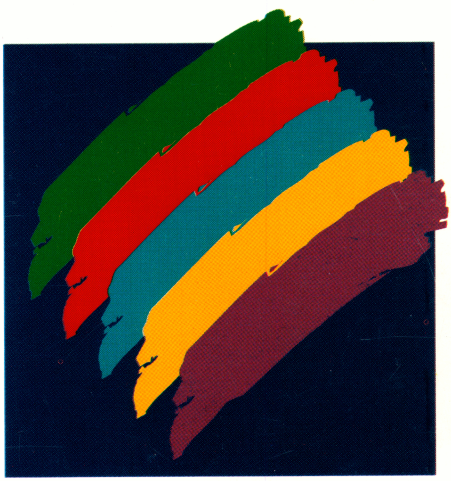
User's Guide

ARexx

AMIGA



Commodore

User's Guide

ARexx

AMIGA

Commodore

COPYRIGHT

Copyright © 1993 by Commodore Electronics Limited. All rights Reserved. This document may not, in whole or in part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Commodore Electronics Limited.
If this product is being acquired for on behalf of the United States of America, its agencies and/or instrumentalities, it is provided with RESTRICTED RIGHTS, and all use, duplication, or disclosure with respect to the included software and documentation is subject to the restrictions set forth in subdivision (b) (3) (ii) of The Rights in Technical Data and Computer Software clause at 252.227-7013 of the DOD FAR. Unless otherwise indicated, the manufacturer/integrator is Commodore Business Machines, Inc., 1200 Wilson Drive, West Chester, PA 19380.

The material set forth in *Using AmigaDOS* is adapted from *The AmigaDOS Manual,* 2nd Edition, Copyright © 1987 by Commodore-Amiga, Inc. used by permission of Bantam Books. All Rights Reserved. The Times Roman, Helvetica Medium, and Courier fonts included in the Fonts directory on the Fonts disk are Copyright © 1985, 1987 Adobe Systems, Inc. The CG Times, Univers Medium, and LetterGothic fonts included on the Fonts disk are Copyright © 1990 by Agfa Corporation and under license from the Agfa Corporation.

DISCLAIMER

With this document Commodore makes no warranties or representations, either expressed or implied, with respect to the products described herein. The information presented herein is being supplied on an "AS IS" basis and is expressly subject to change without notice. The entire risk as to the use of this information is assumed by the user. IN NO EVENT WILL COMMODORE BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY CLAIM ARISING OUT OF THE INFORMATION PRESENTED HEREIN, EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITIES OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE LIMITATION OF IMPLIED WARRANTIES OR DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT APPLY.

TRADEMARKS

Commodore, the Commodore logo, CBM, and AUTOCONFIG are trademarks of Commodore Electronics Limited in the United States and other countries. Amiga, AmigaDOS, Kickstart, Workbench and Bridgeboard are trademarks of Commodore-Amiga, Inc. in the United States and other countries.

MS-DOS is a registered trademark of Microsoft Corporation. Compugraphic, CG, and Intellifont are registered trademarks of Agfa Corp. CG Triumvirate is a trademark of Afga Corp. CG Times is based on Times New Roman under license from The Monotype Corporation plc. Times New Roman is a registered trademark of Monotype Corporation. Univers is a registered trademark of Linotype AG. Universe is under license from Haas Typefoundry Ltd. Diablo is a registered trademark of Xerox Corporation; Epson is a registered trademark of Epson America, Inc.; IBM and Proprinter XL are registered trademarks of International Business Machines Corp; Imagewriter is a trademark of Apple Computer, Inc.; LaserJet and LaserJet PLUS are trademarks of Hewlett-Packard Company; NEC and Pinwriter are registered trademarks of NEC Information Systems; Okidata is a registered trademark of Okidata, a division of Oki America, Inc.; Okimate 20 is a trademark of Okidata, a division of Oki America., Inc. This document may also contain references to other trademarks which are believed to belong to the sources associated therewith.

Printed in Germany, Hong Kong, the Philippines, Singapore, Taiwan and the United Kingdom.

This book was produced using a variety of Commodore systems by Isabelle Vesey and Robert Stephenson Weir.

P/N: 371162-01

Table of Contents

Chapter 1
Introducing ARexx

Who is ARexx For? **1-1**
ARexx on the Amiga **1-2**
ARexx Features **1-3**

Chapter 2
Getting Started

Starting ARexx **2-1**
To Start ARexx Automatically 2-1
To Start ARexx Manually 2-2
About ARexx Programs **2-2**
Running ARexx Programs 2-3
Program Examples **2-4**
Amiga.rexx 2-5
Age.rexx 2-5
Calc.rexx 2-6
Even.rexx 2-7
Square.rexx 2-7
Results.rexx 2-8
Grades.rexx 2-9

Chapter 3
Elements of ARexx

Tokens **3-1**
Comments 3-2
Symbols 3-2
Strings 3-5
Operators 3-6
Arithmetic Operators 3-7
Concatenation Operators 3-9
Comparison Operators 3-10
Logical (Boolean) Operators 3-11
Special Characters 3-12
Clauses **3-13**
Null Clauses 3-14
Label Clauses 3-14
Assignment Clauses 3-14
Instruction Clauses 3-15
Command Clauses 3-15
Expressions **3-15**
The Command Interface **3-18**
The Host Address 3-18
Creating a Macro 3-20
Return Codes 3-22
Command Shells 3-22
The Execution Environment **3-23**
The External Environment 3-24
The Internal Environment 3-24
Resource Tracking 3-25

Chapter 4
Instructions

Syntax **4-1**
Alphabetical Reference **4-2**

Chapter 5
Functions

Invoking a Function **5-1**
Types of Functions **5-2**
Internal Function 5-2
Built-In Functions 5-3
External Function Libraries 5-3
The Library List 5-4
External Function Hosts 5-4
The Search Order **5-5**
The Clip List **5-6**
Built-in Functions — Reference **5-7**
Syntax 5-7
Alphabetical Reference 5-8
Example Program 5-35
REXXSupport.Library Functions **5-38**

Chapter 6
Debugging

Tracing **6-1**
Tracing Output 6-3
Command Inhibition 6-4
Interactive Tracing 6-4
Error Processing 6-5
The External Tracing Flag 6-6
Interrupts **6-6**

Chapter 7
Parsing

Templates **7-1**
Markers 7-2
Targets 7-2
Template Objects 7-2
The Scanning Process **7-4**
Parsing Examples **7-5**
Parsing by Tokenization 7-5
Parsing by Pattern 7-6
Parsing by Positional Markers 7-6
Multiple Templates 7-7

Appendix A
Error Messages

Appendix B
Command Utilities

Glossary

Index

Welcome

ARexx, the Amiga® counterpart of the IBM REXX programming language, provides the freedom to customize your work environment. It is especially useful as a scripting language which allows you to control and modify applications and to direct how they interact with each other.

This manual introduces you to ARexx, tells you how to create ARexx programs, and provides a reference section of ARexx commands.

Chapter 1. Introducing ARexx: This chapter gives an overview of ARexx, how it works on the Amiga, and the basic features of the programming language.

Chapter 2. Getting Started: This chapter tells you where to store your ARexx programs, how to execute an ARexx program, and provides several programming examples.

Chapter 3. Elements of ARexx: This chapter details the rules and concepts that make up the ARexx programming language.

Chapter 4. Instructions: This chapter contains an alphabetical listing of ARexx instructions, which are language statements that dictate an action.

Chapter 5. Functions: This chapter describes the use of functions, which are program statements used by ARexx, and provides an alphabetical listing of the built-in ARexx functions.

Chapter 6. Debugging: This chapter focuses on the source-level debugging features used in the development and testing of programs.

Chapter 7. Parsing: This chapter explains how to extract patterns of information from strings.

Appendix A. Error Messages: This appendix lists the ARexx error messages.

Appendix B. Command Utilities: This appendix lists the ARexx commands that can be run from the Shell.

Glossary. The glossary contains common ARexx terms.

Document Conventions

The following conventions are used in this manual:

KEYWORDS	Keywords are displayed in all uppercase letters, however, the arguments are case-insensitive.
| (vertical bar)	Alternative selections are separated by a vertical bar.
{ } (braces)	Required alternatives are enclosed by braces.
[ ] (brackets)	Optional instruction parts are enclosed in brackets.
<n>	Variables are displayed in angle brackets. Do not enter the angle brackets when entering the variable.
`Courier`	Text apearing in the Courier font represents output from your ARexx programs or other information that displays on your screen.
Key1 + Key2	Key combinations displayed with a plus sign (+) connecting them indicate pressing the keys simultaneously.
Key1, Key2	Key combinations displayed with a comma sign (,) separating them indicate pressing the keys in sequence.
Amiga keys	Two keys on the Amiga keyboard used for special functions. The left Amiga key is to the left of the space bar and is marked with a large solid A. The right Amiga key is to the right of the space bar and is an outlined A.

Sources of Additional Information

Further information on learning and using ARexx can be found in:

Modern Programming Using REXX, by R.P. O'Hara and D.G. Gomberg, Prentice-Hall, 1985

The REXX Language: A Practical Approach to Programming, by M. F. Cowlishaw, Prentice-Hall, 1985.

Programming in REXX, J. Ranade IBM Series, by Charles Daney.

Using ARexx on the Amiga, by Chris Zamara and Nick Sullivan, Abacus, 1991.

Amiga ROM Kernel Reference Manual: Libraries, Third Edition, Addison-Wesley, 1992.

Chapter 1
Introducing ARexx

The ARexx programming language can act as a central hub through which applications — even those created by different companies — can exchange data and commands. For example, using ARexx you can instruct a telecommunications package to dial an electronic bulletin board, download financial data from the bulletin board, and then automatically pass the data to a spreadsheet program for statistical analysis — without any user intervention.

ARexx is an interpreted language that uses ASCII file input. The ARexx interpreter is the RexxMast program, located in the System drawer of Workbench. RexxMast monitors the execution of an ARexx program. If RexxMast finds an error while translating or executing a line, it halts and displays an error message on the screen. This interactive testing is both a learning tool and an aid in debugging programs because it immediately highlights when and where an error has occurred.

Who is ARexx For?

You do not need extensive Amiga experience to use ARexx programs and scripts, but you do need to know how:

- To open a Shell and enter AmigaDOS™ commands
- To use a text editor, such as ED or MEmacs
- To create a User-startup file

However, to change the scripts or create your own ARexx scripts, you should have a basic understanding of both the Amiga Workbench™ and AmigaDOS environments. Experienced Amiga users may find ARexx both easier and more powerful than AmigaDOS. In fact, ARexx can be used to enhance or replace existing AmigaDOS commands and scripts, as well as to create integrated applications.

ARexx on the Amiga

ARexx is supported on all Amiga hardware configurations. Beginning with the release of Amiga Workbench Version 2.0, ARexx has been integrated into the Amiga operating system. Specifically, ARexx uses two important features of the Amiga operating system: multitasking and interprocess communication.

Multitasking is the ability to run more than one program at a time. For example, you can simultaneously edit a file, format a disk, and adjust your screen's colors.

Interprocess communication (IPC) is the ability to allow the exchange of information between applications. Interprocess communication is accomplished through the use of message ports, an address contained in an application that can receive and send messages, attached to each program. Each message port has a name and sending a message to an application requires the use of the port's name in an ARexx script.

The sequence of events in sending and receiving a message is:

1. On initialization an application opens its message port.
2. The application waits to receive a message.
3. The Amiga operating system notifies the application that a message has arrived at its port.
4. The application acts on that message.
5. The application notifies the message's sender (ARexx) that the message has been received and processed.

This transfer of messages is not limited to one application and ARexx. Several applications can send messages back and forth using ARexx as the central transfer location. However, all the applications must be ARexx-compatible.

ARexx Features

Features of the ARexx programming language are:

- Typeless Data — Data is treated as individual character strings and variable values are undeclared.
- Interpreted Execution — The read-and-execute ability of ARexx skips the extra step of program compilation.
- Automatic Resource Management — Automatic internal memory allocation removes unnecessary strings and data.
- Tracing, Trapping, and Debugging — Tracing and trapping permit handling of errors that would normally abort the program. The debugging facilities allow you to view your entire program, reducing development and testing time.
- Function Libraries — External function libraries provide extended, pre-programmed functions.

Chapter 2
Getting Started

This chapter shows you how to:

- Start ARexx
- Save Programs
- Store Programs
- Use sample programs

Starting ARexx

To start using ARexx, you activate the RexxMast program. The RexxMast program is started automatically or manually. Each time ARexx is started or stopped, a text message appears.

To Start ARexx Automatically

There are two methods to start ARexx automatically: placing the RexxMast icon in the WBStartup drawer or editing the S:User-Startup file.

To place RexxMast in the WBStartup drawer:

1. Open the System drawer.
2. Drag the RexxMast icon over the WBStartup drawer.
3. Reboot your Amiga.

To edit the S:User-Startup file:

1. Open a text editor.
2. Open the S:User-Startup file.

3. Enter REXXMAST>NIL:.
4. Save the file.
5. Reboot your Amiga.

To Start ARexx Manually

There are two ways to start RexxMast manually: double-click on the RexxMast icon in Workbench or start it from the Shell. Floppy-based system users can save disk space by starting ARexx only when necessary.

To start RexxMast from Workbench:

1. Open the System Drawer.
2. Double-click on the RexxMast icon.

To start RexxMast from the Shell:

1. Open a Shell.
2. Type REXXMAST >NIL: and press Enter.

About ARexx Programs

ARexx programs are usually stored in the REXX: directory (which is generally assigned to the SYS:S directory). Although programs can be stored in any directory, storing them in REXX: has several advantages:

- You can run the program without having to type the complete path.
- All of your ARexx programs will be in the same place.
- Most applications search for ARexx programs in REXX:.

Just as you can store an ARexx program anywhere, you can also name it anything you choose. However, adopting a simple naming convention will make program management much easier. Programs run from the Shell should have a .rexx extension to distinguish them from files run from other applications.

Running ARexx Programs

The RX command is used to run an ARexx program. If a complete path is included with the program name, only that directory is searched for the program. If no path is included, the current directory and REXX: are checked.

As long as your program is stored in the REXX: directory, you do not need to include the .rexx extension when specifying your program name. In other words, typing:

```
RX Program.rexx
```

is the same as:

```
RX Program
```

A short program can be entered directly at the command line by enclosing the program line in double-quotes. For example, the following program will send five files named myfile.1 through myfile.5 to the printer.

```
RX "DO i=1 to 5;
ADDRESS command 'copy myfile.' || i 'prt:'; END"
```

When an application is ARexx-compatible, you can run ARexx programs from within the application by choosing a menu item or by specifying command options. Refer to the application's documentation for more information.

ARexx programs can be run from the Workbench by creating a tool or project icon for the program. You must specify the RX command as the Default Tool for the icon. In the icon's Information window, enter:

```
Default Tool:  SYS:Rexxc/RX
```

When the icon is opened, RX starts RexxMast (if it is not already running). It executes the file associated with the icon as an ARexx program.

ARexx accepts two Tool Types: Console, to specify a window, and CMD, to specify a command string. You enter these Tool Types in the project icon's Information window as:

```
Console=CON:0/0/640/200/Example/Close
CMD=rexxprogram
```

Program Examples

The following examples illustrate how to use ARexx to display text strings on your screen, to perform calculations, and to activate the error checking feature.

Programs can be entered into any text editor, such as ED or MEmacs, or a word processor. Save your program as an ASCII file if you use a word processor. ARexx supports the extended ASCII character set (Å, Æ, ß). These extended characters are recognized as ordinary printing characters and will be mapped from lowercase to uppercase.

The examples also illustrate the use of some basic ARexx syntax requirements such as:

- Comment lines
- Spacing rules
- Case-sensitivity
- Use of single and double quotes

Each ARexx program consists of a comment line that describes the program and an instruction that displays text on the console. ARexx programs must always begin with a comment line. The initial (slash asterisk) /* balanced with an ending (asterisk slash) */ tells the RexxMast interpreter that it has found an ARexx program. Without the /* and the *\, RexxMast will not view the file as an ARexx program. Once it begins executing the program, ARexx ignores any additional comment lines within the file. However, comment lines are extremely useful when reading the program. They can help organize and make sense of a program.

Amiga.rexx

This program shows how to use SAY in a set of instructions to display text strings on the screen. Instructions are language statements that denote a certain action to be performed. Each statement always begins with a symbol. In the following example, the symbol is SAY. (Symbols are always translated to uppercase letters when the program is run.) Following SAY is an example of a string. A string is a series of characters surrounded by single quotes (') or double quotes (").

Program 1. Amiga.rexx

```
/*A simple program*/
SAY 'Amiga.  The Computer For the Creative Mind.'
```

Enter the above program, and save it as REXX:Amiga.rexx. To run the program, open a Shell window and type:

```
RX Amiga
```

Although the full path and program name is Rexx:Amiga.rexx, you do not need to type the REXX: directory name or the .rexx extension if the program has been saved in the REXX: directory.

You should see the following text in your Shell window:

```
Amiga.  The Computer for the Creative Mind.
```

Age.rexx

This program displays a prompt for input and then reads entered information.

Program 2. Age.rexx

```
/*Calculate age in days*/
SAY 'Please enter your age:'
PULL age
SAY 'You are about' age*365 'days old.'
```

Save this program as REXX:Age.rexx and run it with the command:

```
RX age
```

This program begins with a comment line that describes what the program will do. All ARexx programs begin with a comment. The SAY instruction displays a request for input on the console.

The PULL instruction reads a line of input from the user, which in this case is the user's age. PULL takes the input, converts it to uppercase letters, and stores it in a variable. Variables are symbols which may be assigned a value. Choose descriptive variable names. This example uses the variable name "age" to hold the entered number.

The final line multiplies the variable "age" by 365 and issues the SAY instruction to display the result. The "age" variable did not have to be declared as a number because its value was checked when it was used in the expression. This is an example of typeless data. To see what would happen if age was not a number, try running the program again with a non-numeric entry for the age. The resulting error message shows the line number and type of error that occurred.

Calc.rexx

This program introduces the DO instruction, which repeats the execution of program statements. It also illustrates the exponentiation operator (**). Enter this program and save it as REXX:Calc.rexx. To run the program, use the "RX calc" command.

Program 3. Calc.rexx

```
/*Calculate some squares and cubes.*/
DO i = 1 to 10     /*Begin loop — 10 iterations*/
   SAY i i**2 i**3    /*Perform calculations*/
   END                /*End of loop*/
SAY 'All done.'
```

The DO instruction repeatedly executes the statements between the DO and END instructions. The variable "i" is the index variable for the loop and is incremented by 1 for each iteration (repetition). The number following the symbol TO is the limit for the DO instruction and could have been a variable or a full expression rather than just the constant 10.

Generally, ARexx programs use single spacing between alphanumeric characters. In Program 3, however, spacing is closed

up between the exponentiation characters (**) and the variables (i, and 2, i and 3).

The statements within the loop have been indented. This is not required by the language, but it makes the program more readable, because you can easily visualize where the loop starts and stops.

Even.rexx

The IF instruction allows statements to be conditionally executed. In this example, the numbers from 1 to 10 are classified as odd or even by dividing them by 2 and then checking the remainder. The // arithmetic operator calculates the remainder after a division operation.

Program 4. Even.rexx

```
/*Even or odd?*/
DO i = 1 to 10 /*Begin loop - 10 iterations*/
   IF i // 2 = 0 THEN type = 'even'
     ELSE type = 'odd'
   SAY i 'is' type
   END             /*End loop*/
```

The IF line states that if the remainder of the division of the variable "i" by 2 equals 0, then set the variable "type" to even. If the remainder is not 0, the program will skip over the THEN branch and execute the ELSE branch, setting variable "type" to odd.

Square.rexx

This example introduces the concept of a function, a group of statements executed by mentioning the function name in a suitable context. Functions allow you to build large complex programs from smaller modules. Functions also permit the same code for similar operations in a different program.

Functions are specified in an expression as a name followed by an open parenthesis. (There is no space between the name and the parenthesis.) One or more expressions, called arguments, may follow the parenthesis. The last argument must be followed by a

closing parenthesis. These arguments pass information to the function for processing.

Program 5. Square.rexx

```
/*Defining and calling a function.*/
DO i = 1 to 5
   SAY i square(i) /*Call the "square" function*/
   END
EXIT
square:         /*Function name*/
   ARG x           /*Get the argument*/
   RETURN x**2 /*Square it and return*/
```

Starting with DO and ending with END, a loop is set up with an index variable "i", that will increment by 1. The loop will iterate (repeat) five times. The loop contains an expression that calls the function "square" when the expression is evaluated. The function's result is displayed using the SAY instruction.

The function "square", defined by the ARG and RETURN instructions, calculates the squared values. ARG retrieves the value of the argument string "i" and RETURN passes the function's result back to the SAY instruction.

Once the function is called by the loop, the program looks for the function name "square:", retrieves the argument "i", performs a calculation, and returns to the line within the DO/END loop. The EXIT instruction ends the program after the final loop.

Results.rexx

The TRACE instruction activates ARexx's error checking feature.

Program 6. Results.rexx

```
/*Demonstrate "results" tracing*/
TRACE results
sum = 0 ; sumsq = 0;
DO i = 1 to 5
   sum = sum + 1
   sumsq = sumsq + i**2
   END
SAY 'sum=' sum 'sumsq=' sumsq
```

The console displays the executed source lines, each pass through the DO/END loop, and the expression's final results. Removing the TRACE instruction, would display only the final result: sum = 15 sumsq = 55.

Grades.rexx

This program calculates the final grade for a given student based on four essay grades and a class participation grade. The average of Essay 1 and Essay 2 is worth 30%, the average of Essay 3 and Essay 4 is worth 45%, and participation is worth 25% of the final grade.

Once a final grade is displayed, an option to continue with another calculation is presented. The response is "PULLed" and if it does not equal Q (quit), the loop continues. If the response equals Q, the program quits the loop and exits.

Program 7. Grades.rexx

```
/*Grading program*/
SAY "Hello, I will calculate your grades for you."
response = 0
DO while response ~ = "Q" /*Loop while response isn't Q*/
   SAY "Please enter all grades for the student."
   SAY "Essay 1:"
   PULL es1
   SAY "Essay 2:"
   PULL es2
   SAY "Essay 3:"
   PULL es3
   SAY "Essay 4:"
   PULL es4
   SAY "Participation:"
   PULL p
  Final = (((es1 + es2)/2*.3) + ((es3 + es4)/2*.45) + (p*.25))
   SAY "Your final grade for this student is " Final
   SAY "Would you like to continue? (Q for quit.)"
   PULL response
   END
EXIT
```

Chapter 3
Elements of ARexx

This chapter introduces the rules and concepts that make up the ARexx programming language and explains how ARexx interprets the characters and words used in programs. The different elements that are explained include:

- Tokens — the smallest element of the ARexx language
- Clauses — the smallest executable unit, similar to a sentence
- Expressions — a group of evaluated tokens
- The Command Interface — the process by which ARexx programs communicate with ARexx-compatible applications

This chapter also includes a discussion of the ARexx execution environment. This is intended for more advanced Amiga users and includes technical details on interprocess communication.

Tokens

Tokens, the smallest distinct entities of the ARexx language, may be a single character or a series of characters. There are five categories of tokens:

- comments
- symbols

- strings
- operators
- special characters

Comments

A comment is any group of characters beginning with the sequence /* (slash asterisk) and ending with */ (asterisk slash). Each ARexx program must begin with a comment. Each /* must have a matching */. For example:

```
/*This is an ARexx comment*/
```

Comments may be placed anywhere in a program and can even be nested within one another. For example:

```
/*A /*nested*/ comment*/
```

Insert comments throughout your program. Comments remind you and others of the program's intentions. Because the interpreter ignores comments when it scans your programs, comments do not slow down the execution of your program.

Symbols

A symbol is any group of the characters a-z, A-Z, 0-9, and period (.), exclamation point (!), question mark (?), dollar sign ($), and underscore(_). Symbols are translated to uppercase as the interpreter scans the program, so the symbol MyName is equivalent to MYNAME. The four types of recognized symbols are:

Fixed symbols	A series of numeric characters that begins with a digit (0-9) or a period (.). The value of a fixed symbol is always the symbol name itself, translated to uppercase. 12345 is an example of a fixed symbol.
Simple symbols	A series of alphabetic characters that begins with a letter A-Z. "MyName" is an example of a simple symbol.
Stem symbols	A series of alphanumeric characters that ends with one period. "A." and "Stem9." are examples of stem symbols.
Compound symbols	A series of alphanumeric characters that includes one or more periods within the characters. "A.1.Index" is an example of a compound symbol.

Simple, stem, and compound symbols are called variables and may be assigned a value during the course of the program execution. If a variable has not yet been assigned a value, it is uninitialized. The value for an uninitialized variable is the variable name itself (translated to uppercase, if applicable).

Stems and compound symbols have special properties that make them useful for building arrays and lists. Stem symbols provide a way to initialize a whole class of compound symbols. A compound symbol can be regarded as having the structure stem.n_1.n_2 . . . n_k, where the leading name is a stem symbol and each node, n_1. . . n_k, is a fixed or simple symbol.

When an assignment is made to a stem symbol, it assigns that value to all possible compound symbols derived from the stem. Thus, the value of a compound symbol depends on the prior assignments made to itself or its associated stem.

Whenever a compound symbol appears in a program, its name is expanded by replacing each node with its current value. The value

string may consist of any characters, including embedded blanks, and will not be converted to uppercase. The result of the expansion is a new name that is used in place of the compound symbol. For example, if J has the value 3 and K has the value 7, then the compound symbol A.J.K will expand to A.3.7.

Compound symbols can be regarded as a form of associative or content-addressable memory. For example, suppose that you needed to store and retrieve a set of names and telephone numbers. The conventional approach would be to set up two arrays, NAME and NUMBER, each indexed by an integer running from one to the number of entries. A number would be looked up by scanning the name array until the given name was found, say in NAME.12, and then retrieving NUMBER.12. With compound symbols, the symbol NAME could hold the name to be retrieved, and NUMBER.NAME would then expand to the corresponding number, for example, NUMBER.CBM.

Compound symbols can also be used as conventional indexed arrays, with the added convenience that only a single assignment (to the stem) is required to initialize the entire array.

For instance, the program below uses the stems "number." and "addr." to create a computerized telephone directory.

Program 8. Phone.rexx

```
/*A telephone book to show compound variables.*/
IF ARG() ~ = 1 THEN DO
SAY "USAGE: rx phone name"
   EXIT 5
   END
/*Open window to display phone nos/addresses.*/
CALL OPEN out, "con:0/0/640/60/ARexx Phonebook"
IF ~ result THEN DO
   SAY "Open failure ... sorry"
   EXIT 10
   END
/*Number definitions*/
number. = '(not found)'
number.wsh = '(555) 001-0001'
addr. = '(not found)'
number.CBM = '(555) 002-0002'
addr.CBM = '1200 Wilson Dr., West Chester, PA, 19380'

/*(Work is done here)*/
ARG name   /*The name*/
CALL WRITELN out,name || "'s number is" number.name
CALL WRITELN out,name || "'s address is" addr.name
CALL WRITELN out, "Press Return to exit."
CALL READLN out
EXIT
```

To execute the program, activate a Shell window and enter:

```
RX Phone cbm
```

A window will display the name and address assigned to CBM.

Strings

A string is any group of characters beginning and ending with a quote (') or double quote (") delimiter. The same delimiter must be

used at both ends of the string. To include the delimiter character in the string, use a double-delimiter sequence (`''` or `""`). For example:

"Now is the time."	An example of a normal string.
'Can"t you see?'	An example of a string using a double-delimiter sequence

The value of a string is the string itself. The number of characters in the string is called its length. If the string does not contain any characters, it is called a null string.

Strings that are followed by an X or B character are classified as hex or binary strings, respectively, and must be composed of hexadecimal digits (0-9, A-F) or binary digits (0,1). For example:

```
'4A 3B C0'X
'00110111'B
```

Blanks are permitted at byte boundaries to improve readability. Hex and binary strings are convenient for specifying non-ASCII characters and machine-specific information, like addresses. They are converted immediately to the packed (machine-compressed) internal form.

Operators

Operators are a combination of the following characters: ~ + – * / = > < & | ^, as explained in this section. There are four types of operators:

- Arithmetic operators require one or two numeric operands and produce a numeric result.
- Concatenation operators join two strings into a single string.
- Comparison operators require two operands and produce a Boolean (0 or 1) result.
- Logical operators require one or two Boolean operands and produce a Boolean result.

Each operator has an associated priority that determines the order in which operations will be performed in an expression. Operators with higher priorities (8) are performed before those with lower priorities (1).

Arithmetic Operators

An important class of operands are those representing numbers. Numbers consist of the characters 0-9, a period (.), plus sign (+), minus sign (–), and blanks. To indicate exponential notation, a number may be followed by an "e" or "E" and a (signed) integer.

Both strings and symbols may be used to specify numbers. Since the language is typeless, variables do not have to be declared as numeric before use in an arithmetic operation. Instead, each value string is examined when it is used in order to verify that it represents a number. The following examples are all valid numbers:

```
33
"  12.3  "
0.321e12
' + 15. '
```

Leading and trailing blanks are permitted. Blanks may be embedded between a plus (+) or minus (–) sign and the number, but not within the number itself.

You can modify the basic precision used for arithmetic calculations while a program is executing. The number of significant figures used in arithmetic operations is determined by the Numeric Digits setting and may be modified using the NUMERIC instruction described in Chapter 4.

The number of decimal places used for a result depends on the operation and the number of decimal places in the operands. ARexx preserves trailing zeroes to indicate the precision of the result. If the total number of digits required to express a value exceeds the current Numeric Digits setting, the number is formatted in exponential notation. They are:

- Scientific notation — the exponent is adjusted so that a single digit is placed to the left of the decimal point.
- Engineering notation — the number is scaled so that the exponent is a multiple of 3 and the digits to the left of the decimal point range from 1 to 999.

Table 3-1 lists arithmetic operators.

Table 3-1. Arithmetic Operators

Operator	Priority	Example	Result
+ (prefix conversion)	8	'3.12'	3.12
– (prefix negation)	8	–"3.12"	–3.12
** (exponentiation)	7	0.5 ** 3	0.125
* (multiplication)	6	1.5*1.50	2.250
/ (division)	6	6 / 3	2
% (integer division)	6	–8 % 3	–2
// (remainder)	6	5.1//0.2	0.1
+ (addition)	5	3.1+4.05	7.15
– (subtraction)	5	5.55 – 1	4.55

Concatenation Operators

ARexx defines two concatenation operators. The first, identified by the operator sequence | | (two vertical bars), joins two strings into a single string with no intervening blank. This type of concatenation can also be specified implicitly. When a symbol and a string are typed without any intervening spaces, ARexx behaves as if the | | operator had been specified. The second concatenation operation is identified by the blank operator and joins the two operand strings with one intervening blank.

The priority of all concatenation operations is 4. Table 3-2 summarizes the different operations.

Table 3-2. Concatenation Operators

Operator	Operation	Exampie	Result
||	Concatenation	'why me,'||'Mom?'	why me,Mom?
blank	Blank Concate-nation	'good''times'	good times
none	Implied Concate-nation	one'two'three	ONEtwoTHREE

Comparison Operators

ARexx supports three types of comparisons:

- Exact comparisons — character-by-character comparison.
- String comparisons — ignore leading blanks and add blanks to the shorter string.
- Numeric comparisons — converts the operands to an internal numeric form using the current Numeric Digits setting and then runs an arithmetic comparison.

Comparisons always result in a Boolean value. The numbers 0 and 1 are used to represent the Boolean values false and true. The use of a value other than 0 or 1 when a Boolean operand is expected will generate an error. Any number equivalent to 0 or 1, for example 0.000 or 0.1E1, is also acceptable as a Boolean value.

Except for the exact equality (==) and exact inequality (~==) operators, all comparison operators dynamically determine whether a string or numeric comparison is to be performed. A numeric comparison is performed if both operands are valid numbers. Otherwise, the operands are compared as strings.

All comparisons have a priority of 3. Table 3-3 lists the acceptable comparison operators.

Table 3-3. Comparison Operators

Operator	Operation	Mode
==	Exact equality	Exact
~==	Exact Inequality	Exact
=	Equality	String/Numeric
~=	Inequality	String/Numeric
>	Greater Than	String/Numeric
>= or ~<	Greater Than or Equal To	String/Numeric
<	Less Than	String/Numeric
<= or ~>	Less Than or Equal To	String/Numeric

Logical (Boolean) Operators

ARexx defines the four logical operations, NOT, AND, OR, and Exclusive OR, all of which require Boolean operands and produce a Boolean result. An attempt to perform a logical operation on a non-Boolean operand will generate an error. Table 3-4 shows the acceptable logical operators.

Table 3-4. Logical Operators

Operator	Priority	Operation
~	8	NOT (Inversion)
&	2	AND
|	1	OR
^ or &&	1	Exclusive OR

Special Characters

A few punctuation characters have special meanings within ARexx, as shown in Table 3-5.

Table 3-5. Special Characters

Special Character	Definition
(:) Colon	A colon defines a label when preceded by a symbol token (any alphanumeric character or . ! ? $).
( ) Parentheses	Parentheses are used to group operators and operands into subexpressions to override the normal operator priorities. An open parenthesis also serves to identify a function call within an expression. A Symbol or string followed immediately by an open parenthesis defines a function name. Parentheses must always be balanced within a statement.
(;) Semicolon	A semicolon acts as a statement terminator. Statements too long to fit on one line may be separated by semicolons.

(,) Comma A comma acts as the continuation character for statements broken into several lines and as a separator of argument expressions in a function call.

Clauses

Clauses, the smallest language unit that can be executed as a statement, are formed from token groupings.

As the program is read, the language interpreter splits the program into groups of clauses. These groups of one or more clauses are then broken down into tokens and each clause is classified as a particular type. Seemingly small syntactic differences may completely change the semantic content of a statement. For example:

```
SAY 'Hello, Bill'
```

is an instruction clause and will display "Hello, Bill" on the console, but:

```
''SAY 'Hello, Bill'
```

is a command clause, and will issue "SAY Hello, Bill" as a command to an external program. The presence of the leading null string (' ') changes the classification from an instruction clause to a command clause.

The end of a line normally acts as the implicit end of a clause. A clause can be continued on the next line by ending the line with a comma. The comma is ignored by the program, and the next line is considered as a continuation of the clause. There is no limit to the number of continuations that may occur (except for those limits imposed by the command buffer).

String and comment tokens are automatically continued if a line ends before the closing delimiter has been found, and the newline (i.e., enter) character is not considered to be part of the token.

Null Clauses

Null clauses are lines of blanks or comments and may appear anywhere in a program. They have no function in the execution of a program, except to aid its readability and to increment the line count.

Label Clauses

A label clause is a symbol followed by a colon (:). A label acts as a place marker in the program, but no action occurs with the execution of a label. The colon is considered as an implicit clause terminator, so each label stands as a separate clause. Label clauses may appear anywhere in a program. For example:

```
start:    /*Begin execution*/
syntax:   /*Error processing*/
```

Assignment Clauses

Assignment clauses are identified by a variable symbol followed by an = operator. (In this context the = operator's normal definition of equality comparison is overridden.) The tokens to the right of the = are evaluated as an expression and the result is assigned to the variable. For example:

```
when = 'Now is the time'
answ = 3.14 * fact(5)
```

The equal sign (=) assigns the value 'Now is the time' to the variable 'when', and assigns the result of 3.14 * fact(5) to the variable 'answ'.

Instruction Clauses

Instruction clauses begin with the name of the instruction and tell ARexx to perform an action. Instruction names are described in Chapter 4. For example:

```
DROP a b c
SAY 'please'
IF j > 5 THEN LEAVE;
```

Command Clauses

Command clauses are any ARexx expression that cannot be classified as one of the preceding types of clauses. The expression is evaluated and the result is issued as a command to an external host. For example:

```
'delete' 'myfile'  /*AmigaDOS command*/
'jump' current+10 /*An editor command*/
```

The delete command is not recognized as an ARexx command, so it is sent to the external host, in this case AmigaDOS. The jump command in the second example is assumedly understood by an external text editor.

Expressions

Expressions are a group of evaluated tokens. Most statements contain at least one expression. Expressions are composed of:

- Strings — The value of a string is the string itself.
- Symbols — The value of a fixed symbol is the symbol itself, translated to uppercase. Symbols may be used as variables and may have an assigned value.
- Operators — Operators have a priority order that determines when it will be performed.
- Parentheses — Parentheses may be used to alter the normal order of evaluation in the expression or to identify function calls. A symbol or string followed immediately by an open parenthesis defines the function name, and the tokens between the opening and closing parenthesis form the argument list for the function. For example, the expression:

  ```
  J 'factorial is' fact(J)
  ```

  is composed of:

  - a symbol — J
  - a blank operator
  - a string — factorial is
  - another blank
  - a symbol — fact
  - an open parenthesis
  - a symbol — J
  - a closing parenthesis

In this example, FACT is a function name and (J) is its argument list, the single expression J.

Before the evaluation of an expression proceeds, ARexx must obtain a value for each symbol in the expression. For fixed symbols the value is the symbol name itself, but variable symbols must be looked up in the current symbol table. In the example above, if the symbol

J was assigned the value 3, the expression after symbol resolution would be:

```
3 'factorial is' FACT(3)
```

To avoid ambiguities in the values assigned to symbols during the resolution process, ARexx guarantees a strict left-to-right resolution order. Symbol resolution proceeds irrespective of operator priority or parenthetical grouping. If a function call is found, the resolution is suspended while the function is evaluated. It is possible for the same symbol to have more than one value in an expression.

If the previous example was rearranged to read:

```
FACT(J) 'is' J 'factorial'
```

would the second occurrence of symbol J still resolve to 3? In general, function calls may have side effects that include altering the values of variables. If the example was rearranged, the value of J might have been changed by the call to FACT.

After all symbol values have been resolved, the expression is evaluated based on operator priority and subexpression grouping. ARexx does not guarantee an order of evaluation among operators of equal priority and does not employ a "fast path" evaluation of Boolean operations. For example, in the expression:

```
(1 = 2) & (FACT(3) = 6)
```

the call to the FACT function will be made even though the first term of the AND (&) operation is 0. This example points out that ARexx will continue reading left to right, even though the given example is false and will return a value of 0.

The Command Interface

The ARexx command interface is a public message port. ARexx compatible applications must have this message port. ARexx programs issue commands by placing the command string in a message packet and sending the packet to the host's message port. The program suspends operation while the host processes the commands and resumes when the message packet returns.

The Host Address

ARexx maintains two implicit host addresses, a current and a previous value, as part of the program's storage environment. These values can be changed at any time using the ADDRESS instruction (or its synonym, SHELL). The current host address can be inspected with the ADDRESS() built-in function. The default host address string is REXX, but this can be overridden when a program is invoked. Most host applications will supply the name of their public port when they invoke a macro program, so that the macro can automatically issue commands back to the host.

One special host address is recognized. The string COMMAND indicates that the macro should be issued directly to AmigaDOS. All other host addresses are assumed to refer to a public message port. An attempt to send a command to a nonexistent message port will generate the syntax error "Host environment not found".

Program 9 shows the interaction between ARexx and the AmigaDOS editor, ED. The program sees if ED is running, determines the name of the message port, and sets up some stem variables.

Program 9. ED-status.rexx

```
/*Prints status of ED. ED must be running before this
program is started.  ED ports are named 'Ed', 'Ed_1',
'Ed_2', and so on. */
DEFAULT_ED = "Ed"  /*This name is case sensitive*/
/*Procedure to follow if ED isn't running, or if only
a second (or later) instance of ED is running.*/
DO WHILE ~ SHOW('p',DEFAULT_ED)     /*Look for port*/
SAY "Cannot find port named" DEFAULT_ED
SAY "Available ports:"
SAY SHOW('P') '0a'X
SAY "Enter different name for port, or QUIT to quit"
/* Let user choose port if we can't find it */
DEFAULT_ED = READLN(stdout)
IF STRIP(UPPER(DEFAULT_ED)) = 'QUIT' then exit 10
/*Let user quit*/
END
SAY "Using ED port" DEFAULT_ED
/*Now that port is found, have ARexx address it.*/
ADDRESS VALUE DEFAULT_ED
/*Set up some useful stem variables*/
STEM.0 = 15          /*Number of ED ARexx variables*/
STEM.1 = 'LEFT'                 /*Left margin (SL)*/
STEM.2 = 'RIGHT'               /*Right margin (SR)*/
STEM.3 = 'TABSTOP'          /*Tab stop setting (ST)*/
STEM.4 = 'LMAX'        /*Max visible line on screen*/
STEM.5 = 'WIDTH'         /*Width of screen in chars*/
STEM.6 = 'X'     /*Physical X pos. on screen-from 1*/
STEM.7 = 'Y'     /*Physical Y pos. on screen-from 1*/
STEM.8 = 'BASE'                        /*Window base*/
       /*Base is 0 unless screen is shifted right)*/
STEM.9 = 'EXTEND'      /*Extended margin value (EX)*/
STEM.10 = 'FORCECASE'       /*Case sensitivity flag*/
STEM.11 = 'LINE'              /*Current line number*/
STEM.12 = 'FILENAME'            /*File being edited*/
STEM.13 = 'CURRENT'          /*Text of current line*/
STEM.14 = 'LASTCMD'         /*Last extended command*/
STEM.15 = 'SEARCH'             /*Last search string*/
```

```
/*Ask ED to put values into stem variable 'STEM.'*/
'RV' '/STEM/'    /*RV is an ED command used to send
info from ED to ARexx*/
/*STEM.1 is LEFT, and STEM.LEFT now holds a value
from ED. Here is a way to print that information.*/

DO i = 1 to STEM.0
ED_VAR = STEM.1
SAY STEM.i "=" STEM.ED_VAR /*Print ED variable/value*/
END
```

Creating a Macro

ARexx can be used to write programs for any host application that includes a compatible command interface. Some application programs are designed with an embedded macro language and may include many pre-defined macro commands.

Check your macro program for "shortcut" commands. Some programs may include powerful functions that were implemented specifically for use in macro programs.

The interpretation of the received commands depends entirely on the host application. In the simplest case, the command strings will correspond exactly to commands that could be entered directly by a user. For example, positional control (up/down) commands for a text editor would probably have identical interpretations. Other commands may be valid only when issued from a macro program. A command to simulate a menu operation would probably not be entered by the user. In Program 10, the ARexx program is called by ED to transpose two characters.

Program 10. Transpose.rexx

```
/*Given string '123', if cursor is on 3, macro
converts string to '213'.*/

HOST = ADDRESS()     /*Find out which ED called us*/
ADDRESS VALUE HOST          /*. . . and talk to it.*/
'rv' '/CURR/'      /*Have ED put info in stem CURR*/

/*We'll need two pieces of information:*/
currpos = CURR.X      /*Position of cursor on line*/
currlin = CURR.CURRENT /*Contents of current line*/

IF (currpos >2)         /*Must work on current line*/
THEN currpos = currpos - 1
ELSE DO                      /*Report error and exit*/
'sm /Cursor must be at pos. 2 or more to the right/'
EXIT 10
END

/*Need to reverse the CURRPOSth and CURRPOSth-1 chars
and replace the current line with the new one.*/
DROP CURR. /* STEM variable CURR is no longer needed;
                save some memory */

'd'                  /*Tell ED to delete current line*/
currlin = swapch (currpos,currlin)  /*Swap 2 chars*/
'i /'||currlin||'/'           /*Insert modified line*/
DO i = 1 to currpos   /*Place cursor back at start*/
   'cr'       /*ED's 'cursor right' command*/
   END
EXIT       /*All done*/
/*Function to swap two characters*/
swapch: procedure
PARSE ARG cpos,clin
   chl = substr (clin, cpos, 1)    /*Get character*/
   clin = delstr (clin, cpos, 1)  /*Delete it from
    string*/
   clin = insert (chl,clin,cpos-2,1)  /*Insert to
    create transposition*/
```

```
RETURN clin     /*Return modified string*/
```

To execute this example from ED, press ESC then enter:

```
RX "transpose.rexx"
```

You can also assign this string to a function key.

Return Codes

After it finishes processing a command, the host replies with a return code to indicate the status of the command. The documentation for the host application should describe the possible return codes for each command. These codes can be used to determine whether the operation performed by the command was successful.

This return code is placed in the ARexx special variable RC so that it can be examined by the macro. A value of zero means that the command was successful. A return of a positive integer indicates an error condition. The higher the integer, the more severe the error. The return code allows the macro program to determine whether the command succeeded and to take action if it failed.

Command Shells

Although ARexx was designed to work most effectively with programs that support its specific command interface, it can be used with any command shell program that uses standard I/O mechanisms to obtain its input stream. One way to use ARexx is to create an actual command file on the Ram Disk, then pass it directly to the Shell. Program 11 opens a new Shell to run a standard EXECUTE script.

Program 11. Shell.rexx

```
/*Launch a new Shell*/
ADDRESS command
conwindow = "CON:0/0/640/100/NewOne/Close"

/*Create a command file*/
CALL OPEN out,"ram:temp",write
CALL WRITELN out, 'echo "This is a test"'
CALL CLOSE out

/*Open the new Shell window*/
'newshell' conwindow "ram:temp"
EXIT
```

The Execution Environment

Note The material in this section is intended for advanced Amiga users. This information assumes a working knowledge of the Amiga operating system and a familiarity with the Amiga ROM Kernel Manuals.

The ARexx interpreter, RexxMast, provides a uniform execution environment by running each program as a separate process in the Amiga's multitasking operating system. This allows for a flexible interface between an external host program and RexxMast. The host program can proceed concurrently with its operations or can wait for the interpreted ARexx program to finish. Each ARexx program has both an external and internal environment.

The External Environment

The external environment includes its process structure, input and output streams, and current directory. When each ARexx process is created, it inherits the input and output streams and current directory from its client, the external program that invoked the ARexx program. For example, if an ARexx program was started from a Shell, the ARexx program will inherit the input and output stream and current directory of that Shell. The current directory is used as the starting point in any search for a program or data file. External functions are limited to a maximum of 15 arguments.

The Internal Environment

The internal environment of an ARexx program consists of a static global structure and one or more storage environments. The global data values are fixed (static) at the time the program is invoked. These values include the program source code, static data strings, and argument strings. Once the program is running, these values cannot be changed.

ARexx programs invoked as commands usually have only one argument string, although the command tokenization option may provide more than one. A program invoked as an internal function can have any number of arguments. These arguments persist for the duration of the program.

The storage environment includes the symbol table used for variable values, numeric options, trace options, and host address strings. While the global environment is unique, there may be many storage environments during the course of the program execution. Each time an internal function is called, a new storage environment is activated and initialized. The initial values for most fields are inherited from the previous environment, but values may be

changed afterwards without affecting the caller's environment. The new environment persists until control returns from the function.

Every storage environment includes a symbol table to store the value strings that have been assigned to variables. This symbol table is organized as a two-level binary tree. The primary level stores entries for simple and stem symbols. The secondary level is used for compound symbols. All of the compound symbols associated with a particular stem are stored in one tree, with the entry for the stem being the root of the tree.

Symbols are not entered into the table until an assignment is made to the symbol. Once created, entries at the primary level are never removed, even if the symbol subsequently becomes uninitialized. Secondary trees are released whenever a new assignment is made to the stem associated with that tree.

Resource Tracking

ARexx provides complete tracking for all of the dynamically allocated resources that it uses to execute a program. These resources include memory space, DOS files and related structures, and the message port structure. The tracking system was designed to allow a program to shut down at any point without leaving any resources hanging.

It is possible to go outside of the interpreter's resource tracking net by making calls directly to the Amiga operating system from within an ARexx program. It is the programmer's responsibility to trace and return any resources allocated outside of the ARexx resource tracking system. ARexx provides a special interrupt facility so that a program can retain control after an execution error, perform the required cleanup, and exit.

Chapter 4
Instructions

An instruction clause begins with the name of a particular instruction and tells ARexx to perform a certain action. This chapter provides an alphabetical list of the instructions available in ARexx.

Each instruction keyword may be followed by one or more subkeywords, expressions, or other instruction-specific information. Instruction keywords and subkeywords are recognized only in this specific context. This allows the same keywords to be used in a different context as variables or function names. An instruction keyword cannot be followed by a colon (:) or an equals (=) operator.

Syntax

The syntax for each instruction is shown to the right of the keyword heading. The conventions used in the syntax are shown in Table 4-1:

Table 4-1. Syntax Conventions

Convention	Definition
KEYWORD	All keywords and subkeywords are shown in uppercase letters
| (vertical bar)	Alternative selections are separated by a vertical bar
{ } (braces)	Required alternatives are enclosed by braces
[ ] (brackets)	Optional instruction parts are enclosed in brackets

Note The syntax conventions do not apply to the specifications following the keyword heading. They only apply to the syntax shown to the right of the instruction keyword.

For example, the format for the CALL instruction is:

```
CALL {symbol | string} [expression] [,expression,...]
```

You must supply a symbol or a string as an argument. The vertical bar identifies the alternative selections, and the braces indicate that the use of an argument is required. The specification of an expression is optional, as indicated by the brackets.

Examples are given at the end of the instruction specification. Explanations or evaluations of the examples are shown as ARexx comments /*. . . */.

Alphabetical Reference

This section provides an alphabetical list of ARexx's built-in instructions. The syntax of each instruction is shown to the right of the instruction keyword.

ADDRESS ADDRESS [[symbol | string] | [VALUE][expression]]

This instruction specifies a host address for commands issued by the interpreter. A host address is the name of an application's message port to which ARexx commands are sent. ARexx maintains two host addresses: a current and a previous value. Whenever a new host address is supplied, the previous address is lost and the current address becomes the previous one. These host addresses are part of a program's storage environment and are preserved across internal function calls. The current address can be retrieved with the built-in function ADDRESS().

The ADDRESS keyword alone interchanges the current and previous hosts. Repeated execution will toggle between the two host addresses.

ADDRESS {string | symbol} specifies that the new host address is the string or symbol. The value of the string or symbol is the token

itself. Message port names are case-sensitive. The appropriate syntax for a program command to a message port named MyPort is:

```
ADDRESS 'MyPort'
```

Omit the single quotes around MyPort and ARexx looks for the message port MYPORT and generates an error. The current host address becomes the previous address. An expression specified after a string or symbol is evaluated and the result is issued to the specified host. No changes are made to the current or previous address strings. This provides a convenient way to issue a single command to an external host without disturbing the current host addresses. The return code from the command is treated as it would be from a command clause.

If ADDRESS [VALUE] expression is specified, ARexx uses the result of the expression as the new host address, and the current address becomes the previous address. The VALUE keyword may be omitted if the first token of the expression is not a symbol or string. For example:

```
ADDRESS   /*Swap current and previous address.*/
ADDRESS edit    /*The new host address is EDIT.*/
ADDRESS edit 'top'          /*Move to the top.*/
ADDRESS VALUE edit in /*Compute a new host address.*/
```

ARG ARG [template] [,template ...]

ARG is a shorthand form for the PARSE UPPER ARG instruction. It retrieves one or more of the argument strings available to the program and assigns values to the variables in the template. The number of argument strings available depends on whether the program was invoked as a command or a function. Command invocations normally have only one argument string, but functions may have up to 15. The argument strings are not altered by the ARG instruction. ARG returns uppercase letters. For example:

```
ARG first,second   /*Retrieve arguments*/
```

The structure and processing of templates is described briefly with the PARSE instruction on page 4-13.

BREAK BREAK

The BREAK instruction is used to exit from the range of a DO instruction or from within an INTERPRETed string. It is valid only in these contexts. If used within a DO statement, BREAK exits from the innermost DO statement containing the BREAK. This contrasts with the similar LEAVE instruction, which exits only from an iterative (repeating) DO. For example:

```
DO              /*Begin block*/
   IF i>3 THEN BREAK /*Finished?*/
   a = a + 1
   y.a = name
   END           /*End block*/
```

CALL CALL {symbol | string} [expression] [,expression, ...]

The CALL instruction is used to invoke an internal or external function. The function name is specified by the symbol or string token. Any expressions that follow are evaluated and become the arguments to the called function. The value returned by the function is assigned to the special variable RESULT. It is not an error if a result string is not returned. In this case the variable RESULT is DROPped (becomes uninitialized).

The linkage to the function is established dynamically at the time of the call. ARexx follows a specific search order in attempting to locate the called function. For example:

```
CALL CENTER name,length+4,'+'
```

CENTER is the called function. The expressions will be evaluated and passed as arguments to CENTER.

DO DO [[var=exp] | [exp] [TO exp] [BY exp]] [FOR exp] [FOREVER] [WHILE exp | UNTIL exp]

The DO instruction begins a group of instructions executed as a block. The range of the DO instruction includes all statements up to and including an eventual END instruction.

If no subkeywords follow the DO instruction, the block is executed once. Subkeywords can be used to iterate the block until a termination condition occurs. An iterative DO instruction is sometimes called a loop since ARexx "loops back" to perform the instruction repeatedly. The various parts of the DO instruction are:

- An initializer expression of the form "variable=expression" defines the index variable of the loop. The expression is evaluated when the DO range is first activated and the result is assigned to the index variable. On subsequent iterations an expression of the form "variable = variable + increment" is evaluated, where the increment is the result of the BY expression. If specified, the initializer expression must precede any of the other subkeywords.
- The expression following a BY symbol defines the increment to be added to the index variable in each subsequent iteration. The expression must yield a numeric result, which may be positive or negative and need not be an integer. The default increment is 1.
- The result of the TO expression specifies the upper (or lower) limit for the index variable. At each iteration the index variable is compared to the TO result. If the increment (BY result) is positive and the variable is greater than the limit, the DO instruction terminates and control passes to the statement following the END instruction. The loop also terminates if the increment is negative and the index variable is less than the limit.
- The FOR expression must yield a positive whole number when evaluated and specifies the maximum number of iterations to be performed. The loop terminates when this limit is reached, irrespective of the value of the index variable.
- The initializer BY, TO and FOR expressions are evaluated only when the instruction is first activated, so the increment and limits are fixed throughout the execution. A limit is not required. For example, the instruction "DO i=1" will simply count away forever.
- The FOREVER keyword can be used if an iterative DO instruction is required but no index variable is necessary. The loop will be terminated by a LEAVE or BREAK instruction contained within the loop.

- The WHILE expression is evaluated at the beginning of each iteration and must result in a Boolean value. The iteration proceeds if the result is 1 (true); otherwise, the loop terminates.
- The UNTIL expression is evaluated at the end of each iteration and must result in a Boolean value. The instruction continues with the next iteration if the result is 0 (false), and terminates otherwise. (WHILE and UNTIL are mutually exclusive.)

Program 12. Iteration.rexx

```
/*Examples of DO*/
LIMIT = 20; number = 1
DO i=1 to LIMIT for 10 WHILE number < 20
   number = i * number
   SAY "Iteration" i "number=" number
   END
number = number/3.345; i = 0
DO number for LIMIT/5
   i = i + 1
   SAY "Iteration" i "number=" number
   END
```

The output is shown with comment lines for explanation. The comments would not appear on your screen.

```
Iteration 1 number = 1    /*1 * 1 = 1*/
Iteration 2 number = 2    /*2 * 1 = 2*/
Iteration 3 number = 6    /*3 * 2 = 6*/
Iteration 4 number = 24      /*4 * 6 = 24*/
Iteration 1 number = 7.17488789  /*24/3.345 =
7.17488789*/
Iteration 2 number = 7.17488789  /*number doesn't
change*/
Iteration 3 number = 7.17488789  /*limit/5 = 20/5 =
4*/
Iteration 4 number = 7.17488789  /*operation repeats
4 times*/
```

Note If a FOR limit is also present, the initial expression is still evaluated, but the result need not be a positive integer.

DROP DROP variable [variable ...]

The specified variable symbols are reset to their uninitialized state, in which the value of the variable is the variable name itself. It is not an error to DROP a variable that is already uninitialized. DROPping a stem symbol is equivalent to DROPping the values of all possible compound symbols derived from that stem. For example:

```
a = 123    /*Assign a value to a */
DROP a b   /*DROP (remove) the values from A and B*/
SAY a b    /*Results in A B.*/
```

ECHO ECHO [expression]

The ECHO instruction is a synonym for the SAY instruction. It displays the expression result on the console. For example:

```
ECHO "You don't say"
```

ELSE ELSE [;] [conditional statement]

The ELSE instruction provides the alternative conditional branch for an IF statement. It is valid only within the range of an IF instruction and must follow the conditional statement of the THEN branch. If the THEN branch isn't executed, the statement following the ELSE clause is performed.

ELSE clauses always bind to the nearest, preceeding IF statement. It may be necessary to provide "dummy" ELSE clauses for the inner IF ranges of a compound IF statement to allow alternative branches for the outer IF statements. It is not sufficient to follow the ELSE with a semicolon or a null clause. Instead, the NOP (no-operation) instruction can be used. For example:

```
IF i > 2 THEN SAY 'Really?'
   ELSE SAY 'I thought so'
```

END END [variable]

The END instruction terminates the range of a DO or SELECT instruction. If the optional variable symbol is supplied, it is compared to the index variable of the DO statement (which must be iterative). An error is generated if the symbols do not match. For example:

```
DO i=1 to 5    /*Index variable is i*/
   SAY i
   END i       /*End "i" loop*/
```

EXIT EXIT [expression]

The EXIT instruction terminates the execution of a program. It is valid anywhere within a program. The evaluated expression is passed back to the caller as the function or command result.

The processing of the EXIT result depends on whether a result string was requested by the calling program and whether the current invocation resulted from a command or function call.

- If a result string was requested, the expression result is copied to a block of allocated memory and a pointer to the block is returned as the secondary result of the call.
- If the caller did not request a result string and the program was invoked as a command, an attempt is made to convert the expression result to an integer. This value is then returned as the primary result, with 0 as the secondary result. This allows the EXIT information to be interpreted as a return code by the caller.

For example:

```
EXIT           /*No result needed*/
EXIT 10        /*An error return*/
```

IF IF expression [THEN] [;] [conditional statement]

The IF instruction is used in conjunction with THEN and ELSE instructions to conditionally execute a statement. The result of the expression must be a Boolean value. If the result is 1 (True), the statement following the THEN symbol is executed. Otherwise, control passes to the next statement. The THEN keyword need not immediately follow the IF expression, but may appear as a separate clause.

The instruction is analyzed as "IF expression; THEN; statement". The expression following the IF statement establishes the test condition that determines whether subsequent THEN or ELSE clauses will be performed. Any valid statement may follow the THEN symbol. In particular, a "DO . . . END;" group allows a series of statements to be performed conditionally. For example:

```
IF result < 0 THEN exit        /*All done?*/
```

INTERPRET INTERPRET expression

The INTERPRET command treats the expression as though it were a source statement block. The expression is evaluated and the result is executed as one or more program statements. The statements are considered as a group, as if surrounded by a "DO . . . END" combination. Any statements can be included in the INTERPRETed source, including DO or SELECT instructions. The BREAK instruction can be used to terminate the processing of INTERPRETed statements.

An INTERPRET instruction activates a control range when it is executed, which serves as a boundary for LEAVE and ITERATE instructions. These instructions can only be used with DO loops defined within the INTERPRET. While it is not an error to include label clauses within the interpreted string, only those labels defined in the original program are searched during a transfer of control.

The INTERPRET instruction can be used to construct programs dynamically and then execute them. Program fragments may be passed as arguments to functions, which then INTERPRET the fragments. For example:

```
inst = 'SAY'              /*An instruction*/
INTERPRET inst hello      /*. . . "SAY HELLO"*/
```

ITERATE ITERATE [variable]

The ITERATE instruction terminates the current iteration of a DO instruction and begins the next iteration. Effectively, control passes to the END statement and then (depending on the outcome of the UNTIL expression) back to the DO statement. The instruction normally acts on the innermost iterative DO range. An error results if the ITERATE instruction is not contained within an iterative DO instruction.

If several nested ranges exist, the optional variable symbol specifies which DO range is to be exited. The variable is taken as a literal and must match the index variable of a currently active DO instruction. An error results if a matching DO instruction is not found. For example:

```
DO i=1 to 5
   IF i = 3 THEN ITERATE i
   SAY i
   END
```

LEAVE LEAVE [variable]

LEAVE forces an immediate exit from the iterative DO range containing the instruction. An error results if the LEAVE instruction is not contained within an iterative DO instruction. If several nested ranges exist, the optional variable symbol specifies which DO range is to be exited. The variable is taken as a literal and must match the index variable of a currently active DO instruction. An error results if a matching DO instruction is not found. For example:

```
DO i = 1 to limit
   IF i > 5 THEN LEAVE  /*Maximum iterations*/
   END
```

NOP

NOP

The NOP (NO-oPeration) instruction is provided to control the binding of ELSE clauses in compound IF statements. For example:

```
IF i = j THEN            /*First (outer) IF*/
   IF j = k THEN a = 0   /*Inner IF*/
      ELSE NOP           /*Binds to inner IF*/
   ELSE a = a + 1        /*Binds to outer IF*/
```

NUMERIC

NUMERIC {DIGITS | FUZZ} expression NUMERIC FORM {SCIENTIFIC | ENGINEERING}

- The NUMERIC instruction sets options relating to numeric precision and format. The numeric options are preserved when an internal function is called.
- The DIGITS expression option specifies the number of digits of precision for arithmetic calculations. The expression must evaluate to a positive whole number.
- The FUZZ expression option specifies the number of digits to be ignored in numeric comparison operations. This must be a positive whole number, less than the current DIGITS setting.
- The FORM SCIENTIFIC option specifies that numbers that require exponential notation be expressed in scientific notation. The exponent is adjusted so that the mantissa for non-zero numbers is between 1 and 10. This is the default format.
- The FORM ENGINEERING option selects engineering format for numbers that require exponential notation. Engineering format normalizes a number so that its exponent is a multiple of three and the mantissa (if not 0) is between 1 and 1000.

```
NUMERIC DIGITS 12   /*12 digits of precision*/
NUMERIC FORM SCIENTIFIC /*Result in scientific
    notation*/
```

OPTIONS OPTIONS [FAILAT expression]
OPTIONS [PROMPT expression]
OPTIONS [RESULTS]
OPTIONS [CACHE]

The OPTIONS instruction is used to set various internal defaults. The FAILAT expression sets the limit at or above which command return codes will be signaled as errors. It must evaluate to an integer value. The PROMPT expression provides a string to be used as the prompt with the PULL (or PARSE PULL) instruction. The RESULTS keyword indicates that the interpreter should request a result string when it issues commands to an external host.

The internal options controlled by this instruction are preserved across function calls, so an OPTIONS instruction can be issued within an internal function without affecting the caller's environment. If no keyword is specified with the OPTIONS instruction, all controlled options revert to their default settings. The OPTIONS instruction also accepts a NO keyword to reset a selected option to its default value, making it more convenient to reset the RESULTS attribute for a single command without having to reset the FAILAT and PROMPT options.

OPTIONS also accepts a CACHE keyword that can be used to enable or disable an internal statement-caching scheme. The cache is normally enabled. For example:

```
OPTIONS FAILAT 10
OPTIONS PROMPT "Yes Boss?"
OPTIONS RESULTS
```

OTHERWISE OTHERWISE [;] [conditional statement]

This instruction is valid only within the range of a SELECT instruction and must follow all of the "WHEN . . . THEN" statements. If none of the preceding WHEN clauses have succeeded, the statement following the OTHERWISE instruction is executed. An OTHERWISE is not mandatory within a SELECT range. However, an error will result if the OTHERWISE clause is omitted and none of the WHEN instructions succeed. For example:

```
SELECT
   WHEN i=1 THEN say 'one'
   WHEN i=2 THEN say 'two'
   OTHERWISE SAY 'other'
   END
```

PARSE PARSE [UPPER] inputsource [template] [,template ...]

The PARSE instruction provides a mechanism to extract one or more substrings from a string and assign them to variables. The input string can come from a variety of sources, including argument strings, an expression, or from the console.

Parsing is controlled by a template, which may consist of symbols, strings, operators and parentheses. The template provides both the variables to be given values and the way to determine the value strings. During the parsing operation the input string is split into substrings that are assigned to the variable symbols in the template. The process continues until all of the variables in the template have been assigned a value. If the input string is "used up", any remaining variables are given null values.

When a variable in the template is followed immediately by another variable, the value string is determined by breaking the input string into words separated by blanks. Leading and trailing blanks are not permitted. Each word is assigned to a variable in the template. Normally the last variable receives the untokenized remainder of the input string, since it is not followed by a symbol. A placeholder symbol, a period (.), forces the variable with the period to terminate at the first space in the input stream. Placeholders behave like variables except that they are never assigned a value.

The template may be omitted if the instruction is intended only to create the input string. Templates are described in Chapter 7.

The goal of the parsing operation is to associate a current and next position with each variable symbol in the template. The substring between these positions is then assigned as the value to the variable.

The different options of the instruction are described below.

- The optional UPPER keyword may be used with any of the input sources and specifies that the input string is to be translated to uppercase before being parsed. It must be the first token following PARSE.
- The sources for the input strings are specified by the keyword symbols explained below. When multiple templates are supplied, each template receives a new input string, although for some source options the new string will be identical to the previous one. The input source string is copied before being parsed, so the original strings are never altered by the parsing process.
  - The ARG input option retrieves the argument strings supplied when the program was invoked. Command invocations normally have only a single argument string, but functions may have up to 15 argument strings.
  - The EXTERNAL input string is read from STDERR stream, (see Chapter 6) so as not to disturb any PUSHed or QUEUEd data. If multiple templates are supplied, each template will read a new string. This source option is the same as PULL.
  - The NUMERIC input option places the current numeric options in a string in the order DIGITS, FUZZ and FORM, separated by a single space.
  - The PULL input option reads a string from the input console. If multiple templates are supplied, each template will read a new string.
  - The SOURCE input option retrieves the "source" string for the program. This string is formatted as:

    ```
    {COMMAND|FUNCTION} {0|1} CALLED RESOLVED EXT HOST
    ```

    where:

    - {COMMAND | FUNCTION} indicates whether the program was invoked as a command or as a function.
    - {0 | 1} is a Boolean flag indicating whether a result string was requested by the caller.
    - CALLED is the name used to invoke this program.
    - RESOLVED is the final resolved name of the program.
    - EXT is the file extension to be used for searching (the default is "REXX").

- HOST is the initial host address for commands.

The SOURCE option now returns the full path name of the ARexx program file. Formerly just a relative name was given, which was not sufficient to locate the program's source file.

The "VALUE expression WITH" input string is the result of the supplied expression. The WITH keyword is required to separate the expression from the template. The expression result may be parsed repeatedly by using multiple templates, but the expression is not re-evaluated.

The "VAR variable" input option uses the value of the specified variable as the input string. When multiple templates are provided, each template uses the current value of the variable. This value may change if the variable is included as an assignment target in any of the templates.

The VERSION input option of the current configuration of the ARexx interpreter is supplied in the form:

```
ARexx VERSION CPU MPU VIDEO FREQ
```

where:

- VERSION is the release level of the interpreter, formatted as 1.14.
- CPU indicates the processor currently running the program, and will be one of the values 68000, 68010, 68020, 68030 or 68040.
- MPU will be either NONE, 68881, or 68882, depending on whether a math coprocessor is available.
- VIDEO will indicate either NTSC or PAL.
- FREQ gives the clock (line) frequency as either 60Hz or 50Hz.

For example:

```
/*Numeric string is: "9 0 SCIENTIFIC"*/
PARSE NUMERIC DIGITS FUZZ FORM .
SAY digits     /*9*/
SAY fuzz       /*0*/
SAY form       /*SCIENTIFIC*/
myvar = 1234567890
PARSE VAR myvar 1 a 3 b +2 c 1 d
SAY a
SAY b
SAY c
SAY d
```

This is the output:

```
12
34
567890
1234567890
```

PROCEDURE PROCEDURE [EXPOSE variable [variable...]]

The PROCEDURE instruction is used within an internal function to create a new symbol table. This protects the symbols defined in the caller's environment from being altered by the execution of the function. PROCEDURE is usually the first statement within the function, although it is valid anywhere within the function body. It is an error to execute two PROCEDURE statements within the same function.

The EXPOSE subkeyword provides a selective mechanism for accessing the caller's symbol table, and for passing global variables to a function. The variables following the EXPOSE keyword are taken to refer to symbols in the caller's table. Any subsequent changes made to these variables will be reflected in the caller's environment.

The variables in the EXPOSE list may include stem or compound symbols, in which case the ordering of the variables becomes significant. The EXPOSE list is processed from left to right and compound symbols are expanded based on the values in effect in the new generation. For example, suppose that the value of the symbol J in the previous generation is 123, and that J is uninitialized in the

new generation. Then PROCEDURE EXPOSE J A.J will expose J and A.123, whereas PROCEDURE EXPOSE A.J J will expose A.J and J. Exposing a stem has the effect of exposing all possible compound symbols derived from that stem. That is, PROCEDURE EXPOSE A. exposes A.I, A.J, A.J.J, A.123, etc. For example:

```
fact: PROCEDURE       /*A recursive function*/
   ARG i
   IF i = 1
   THEN RETURN 1
   ELSE RETURN i * fact(i-1)
```

PULL PULL [template] [,template...]

Pull is the shorthand form of the PARSE UPPER PULL instruction. It reads a string from the input console, translates it to uppercase and parses it using the template. Multiple strings can be read by supplying additional templates. The instruction will read from the console even if no template is given. (Templates are described in Chapter 7.) For example:

```
PULL first last .      /*Read names*/
```

PUSH PUSH [expression]

The PUSH instruction is used to prepare a stream of data to be read by a command shell or other program. It appends a newline to the result of the expression then stacks or "pushes" it into the STDIN stream. Stacked lines are placed in the stream in "last-in, first-out" order and are available to be read just as though they had been entered interactively. For example, after issuing the instructions:

```
PUSH line 1
PUSH line 2
PUSH line 3
```

the stream would be read in the order line 3, line 2, and line 1.

PUSH allows the STDIN stream to be used as a private scratch pad to prepare data for subsequent processing. For example, several files could be concatenated with delimiters between them by simply

reading the input files, PUSHing the line into the stream and inserting a delimiter where required. For example:

```
DO i=1 to 5
   PUSH 'echo "Line 'i'"'
    END
```

QUEUE QUEUE [expression]

The QUEUE instruction is used to prepare a stream of data to be read by a command shell or other program. It is very similar to the PUSH instruction and differs only in that the data lines are placed in the STDIN stream in "first-in, first-out" order. In this case, the instructions:

```
QUEUE line 1
QUEUE line 2
QUEUE line 3
```

would be read in the order line 1, line 2, and line 3. The QUEUEd lines always precede all interactively-entered lines and always follow any PUSHed (stacked) lines. For example:

```
DO i=1 to 5
   QUEUE 'echo "Line 'i'"'
   END
```

RETURN RETURN [expression]

RETURN is used to leave a function and return control to the point of the previous function invocation. The evaluated expression is returned as the function result. If an expression is not supplied, an error may result in the caller's environment. Functions called from within an expression must return a result string and will generate an error if no result is available. Functions invoked by the CALL instruction need not return a result.

A RETURN issued from the base environment of a program is not an error and is equivalent to an EXIT instruction. Refer to the EXIT instruction for a description of how result strings are passed back to an external caller. For example:

```
RETURN 6*7    /*Returns 42*/
```

SAY SAY [expression]

The result of the evaluated expression is written to the output console, with a newline character appended. If the expression is omitted, a null string is sent to the console. For example:

```
SAY 'The answer is ' value
```

SELECT SELECT

SELECT begins a group of instructions containing one or more WHEN clauses and possibly a single OTHERWISE clause, each followed by a conditional statement. Only one of the conditional statements within the SELECT group will be executed. Each WHEN statement is executed in succession until one succeeds. If none succeeds, the OTHERWISE statement is executed. The SELECT range must be terminated by an END statement. For example:

```
SELECT
   WHEN i=1 THEN SAY 'one'
   WHEN i=2 THEN SAY 'two'
   OTHERWISE SAY 'other'
   END
```

SHELL SHELL [symbol | string] | [[VALUE] [expression]]

The SHELL instruction is a synonym for the ADDRESS instruction. For example:

```
SHELL edit    /*Set host to 'EDIT'*/
```

SIGNAL SIGNAL {ON | OFF} condition SIGNAL [VALUE] expression

SIGNAL {ON | OFF} controls the state of the internal interrupt flags. Interrupts allow a program to detect and retain control when certain errors occur. In this form SIGNAL must be followed by one of the keywords ON or OFF and one of the condition keywords listed below. The interrupt flag specified by the condition symbol is then set to the indicated state. The valid signal conditions are:

BREAK_C	A Ctrl+C break was detected.
BREAK_D	A Ctrl+D break was detected.
BREAK_E	A Ctrl+E break was detected.
BREAK_F	A Ctrl+F break was detected.
ERROR	A host command returned a non-zero code.
HALT	An external HALT request was detected.
IOERR	An error was detected by the I/O system.
NOVALUE	An uninitialized variable was used.
SYNTAX	A syntax or execution error was detected.

The condition keywords are interpreted as labels to which control will be transferred if the selected condition occurs. For example, if the ERROR interrupt is enabled and a command returns a non-zero code, ARexx will transfer control to the label ERROR:. The condition label must be defined in the program; otherwise, an immediate SYNTAX error results and the program exits.

In SIGNAL [VALUE] expression, the tokens following SIGNAL are evaluated as an expression. An immediate interrupt is generated that transfers control to the label specified by the expression result. The instruction thus acts as a "computed goto".

Whenever an interrupt occurs, all currently active control ranges (IF, DO, SELECT, INTERPRET, or interactive TRACE) are dismantled before the transfer of control. Thus, the transfer cannot be used to jump into the range of a DO loop or other control structure. Only the control structures in the current environment are affected by a SIGNAL condition, making it safe to SIGNAL from within an internal function without affecting the state of the caller's environment.

The special variable SIGL is set to the current line number whenever a transfer of control occurs. The program can inspect SIGL to determine which line was being executed before the transfer. If an ERROR or SYNTAX condition causes an interrupt, the special variable RC is set to the error code that triggered the interrupt. For the ERROR condition, this code is usually an error severity level. Refer to Appendix A for further details on error codes and severity levels. The SYNTAX condition will always indicate an ARexx error code. For example:

```
SIGNAL on error      /*Enable interrupt*/
SIGNAL off syntax    /*Disable SYNTAX*/
SIGNAL start         /*Goto START*/
```

WHEN WHEN expression [THEN [;] [conditional statement]]

The WHEN instruction is similar to the IF instruction, but is valid only within a SELECT range. Each WHEN expression is evaluated in turn and must result in a Boolean value. If the result is a 1, the conditional statement is executed and control passes to the END statement that terminates the SELECT. As with the IF instruction, the THEN need not be part of the same clause. For example:

```
SELECT
   WHEN i<j THEN SAY 'less'
   WHEN i=j THEN SAY 'equal'
   OTHERWISE  SAY 'greater'
   END
```

Chapter 5
Functions

A function is a program or group of statements that is executed whenever that function name is called in a particular context. A function may be part of an internal program, part of a library, or a separate external program. Functions are an important building block of modular programming because they allow you to construct large programs from a series of small, easily developed modules.

This chapter explains the different types of functions and how they are evaluated. It also provides an alphabetical reference of ARexx's built-in function library.

Invoking a Function

Within an ARexx program, a function is defined as a symbol or string followed immediately by an open parenthesis. The symbol or string (taken as a literal) specifies the function name, and the open parenthesis begins the argument list. Between the opening and closing parentheses are zero or more argument expressions, separated by commas, that supply the data being passed to the function.

Valid function calls are:

```
CENTER ('title',20)
ADDRESS()
'ALLOCMEM'(256*4,1)
```

Each argument expression is evaluated in turn and the resulting strings are passed as the argument list to the function. Each argument expression, while often just a single literal value, can

include arithmetic or string operations or even other function calls. Argument expressions are evaluated from left to right.

Functions can also be invoked using the CALL instruction. The CALL instruction, described in Chapter 4, can be used to invoke a function that may not return a value.

Types of Functions

There are three types of functions:

- Internal functions — defined within the ARexx program.
- Built-in functions — supplied by the ARexx programming language.
- Function Libraries — a special Amiga shared library.

Internal Functions

An internal function is identified by a label within the program. When the internal function is called, ARexx creates a new storage environment so that the previous caller's environment is preserved. The new environment inherits the values from its predecessor, but subsequent changes to the environment variables do not affect the previous environment.

The specific values preserved are:

- The current and previous host addresses.
- The NUMERIC DIGITS, FUZZ, and FORM settings.
- The trace option, inhibit flag, and interactive flag.
- The state of the interrupt flags as defined by the SIGNAL instruction.
- The current prompt string as set by the OPTIONS PROMPT instruction.

The new environment does not automatically get a new symbol table, so initially all of the variables in the previous environment are available to the called function. The PROCEDURE instruction can be used to create a table and thereby protect the caller's symbol

values. PROCEDURE can also be used to allow the same variable name to be used in two different areas with two different values.

Execution of the internal function proceeds until a RETURN instruction is executed. At this point the new environment is dismantled, and control resumes at the point of the function call. The expression supplied with the RETURN instruction is evaluated and passed back to the caller as the function result.

Built-In Functions

ARexx provides a substantial library of predefined functions as part of the language system. These functions are always available and have been optimized to work with the internal data structures. In general, the built-in functions execute much faster than an equivalent interpreted function, so their usage is strongly recommended.

Several of the built-in functions create and manipulate external AmigaDOS files. Files are referenced by a logical name, a case-sensitive name that is assigned to a file when it is first opened. The initial input and output streams are given the names STDIN (standard input) and STDOUT (standard output). There is no theoretical limit to the number of files that may be open simultaneously, although a limit will be imposed by available memory. All open files are closed automatically when the program exits.

External Function Libraries

A function library is a collection of one or more functions organized as an Amiga shared library. The library must reside in LIBS:, but may be either memory or disk-resident. Disk-resident libraries are loaded and opened as needed.

The library has to be especially tailored for use by ARexx. Each function library must contain a library name, a search priority, an entry point offset, and a version number. When ARexx is searching for a function, the interpreter opens each library and checks its "query" entry point. This entry point must be specified as an integer offset (e.g. "–30") from the library base. The return code

from the query call indicates whether the desired function was found. If the function is found, it is called with the parameters passed by the interpreter, and the function result is returned to the caller. If it is not found, a "Function not found" error code is returned, and the search continues with the next library in the list. Function libraries are always closed after being checked so that the operating system can reclaim the memory space if required.

The Library List

The ARexx resident process maintains a list of the currently available function libraries and function hosts called the Library List. Application programs can add or remove function libraries as required.

The Library List is maintained as a priority-sorted queue. Each entry has an associated search priority in the range 100 (highest) to –100 (lowest). Entries can be added at an appropriate priority to control the function name resolution. Libraries with higher priorities are searched first. Within a given priority level, those libraries added first are searched first. The priority levels are significant if any of the libraries have duplicate function name definitions, since the function located further down the search chain could never be called.

External Function Hosts

The name associated with a function host is the name of its public message port. Function calls are passed to the host as a message packet; it is then up to the individual host to determine whether the specified function name is one that it recognizes. The name resolution is completely internal to the host, so function hosts provide a natural gateway mechanism for implementing remote procedure calls to other machines in a network. The ARexx resident process is a function host and is installed in the Library List with a priority of –60.

The Search Order

Function linkages in ARexx are established at the time of the function call. A specific search order is followed until a function matching the name symbol or string is found. If the specified function cannot be located, an error is generated and the expression evaluation is terminated. The full search order is:

Internal Functions — The program source is examined for a label that matches the function name. If a match is found, a new storage environment is created and control is transferred to the label.

Built-In Functions — The built-in function library is searched for the specified name. All of these functions are defined by uppercase names.

Function Libraries and Function Hosts — The available function libraries and function hosts are maintained in the Library List, which is searched starting at the highest priority until the requested function is found or the end of the list is reached. Function hosts are called using a message-passing protocol similar to that used for commands and may be used as gateways for remote procedure calls to other machines in a network.

External ARexx Programs — The final search step is to check for an external ARexx program file by sending an invocation message to the ARexx resident process. The search always begins in the current directory and follows the same search path as the original ARexx program invocation. The name matching process is not case-sensitive.

The function name-matching procedure may be case-sensitive for some of the search steps, but not for others. The matching procedure used in a function library or function host is design dependent. Functions defined with mixed-case names must be called using a string token, since symbol names are always translated to uppercase.

The full search order is followed whenever the function name is defined by a symbol token. However, the search for internal functions is bypassed if the name is specified by a string token.

This allows internal functions to usurp the names of external functions, as in the following example:

```
CENTER:             /*internal "CENTER"*/
ARG string,length      /*get arguments*/
length = MIN(length,60) /*compute length*/
return 'CENTER'(string, length)
```

Here the built-in function CENTER() has been replaced by an internal function after modifying the length argument.

The Clip List

The Clip List is a publicly accessible mechanism that can be used as a general clipboard for intertask communication. Many functions use the clipboard for retrieving different types of information, such as predefined constants or strings.

The Clip List maintains a set of (name, value) pairs that may be used for a variety of purposes. (SETCLIP() is used to add pairs to the list.) Each entry in the list consists of a name and a value string and may be located by name. In general, the names used should be chosen to be unique to an application to prevent unintended duplications with other programs. Any number of entries may be posted to the list.

One potential application for the Clip List is as a mechanism for loading predefined constants into an ARexx program. For example:

```
pi=3.14159; e=2.718; sqrt2=1.414
```

(i.e., a series of assignments separated by semicolons). In use, such a string could be retrieved by name using the built-in function GETCLIP() and then INTERPRETed within the program. The assignment statements within the string would then create the required constant definitions. For example:

```
/*Assume a string called "numbers" is available*/
numbers = GETCLIP('numbers')
INTERPRET numbers        /*. . . assignments*/
```

The strings would not be restricted to contain only assignment statements, but could include any valid ARexx statements. The Clip

List could thus provide a series of programs for initializations or other processing tasks.

The resident process supports addition and deletion operations for maintaining the Clip List. The names in the (name,value) pairs are assumed to be in mixed case and are maintained to be unique in the list. An attempt to add a string with an existing name will simply update the value string. The name and value strings are copied when an entry is posted to the list, so the program that adds an entry is not required to maintain the strings.

Entries posted to the Clip List remain available until explicitly removed. The Clip List is automatically released when the resident process exits.

Built-in Functions — Reference

This section provides an alphabetical list of the built-in functions. The syntax of each function is shown to the right of the function keyword.

Syntax

Optional arguments are shown in brackets and generally have a default value that is used if the argument is omitted. When an option keyword is specified as an argument, only the first character is significant. Option keywords are not case-sensitive.

Many functions accept a pad character argument. Pad characters are inserted to fill or create spaces. For functions that accept a pad argument, only the first character of the argument string is significant. If a null string is supplied, the default padding character, usually a blank, will be used.

In the following examples, an arrow (→) is used as an abbreviation for "evaluates as." The arrow will not be displayed when a program is run. For example:

```
SAY ABS(-5.35) → 5.35
```

This means that SAY ABS(–5.35) is evaluated as 5.35.

Alphabetical Reference

ABBREV() ABBREV(string1,string2[,length])

Returns a Boolean value that indicates whether string2 is an abbreviation of string1 with length greater than or equal to the specified length argument. The default length is 0, so the null string is an acceptable abbreviation. For example:

```
SAY ABBREV('fullname','ful')    → 1
SAY ABBREV('almost','alm',4)    → 0
SAY ABBREV('any','')    →1
```

ABS() ABS(number)

Returns the absolute value of the number argument. This value must be numeric. For example:

```
SAY ABS(-5.35)    → 5.35
SAY ABS(10)    → 10
```

ADDLIB() ADDLIB(name,priority[,offset,version])

Adds a function library or a function host to the library list maintained by the resident process. The name argument specifies either the name of a function library or the public message port associated with a function host. The name is case-sensitive. Any specified libraries should reside in the system LIBS: directory.

The priority argument specifies the search priority and must be an integer between 100 and -100, inclusive. The offset and version arguments apply only to libraries. The offset is the integer offset to the library's "query" entry point, and the version is an integer specifying the minimum acceptable release level of the library.

The function returns a Boolean result that indicates whether the operation was successful. If a library is specified, it is not actually opened at this time. Similarly, ARexx does not check to see whether a specified function host port is open. For example:

```
SAY ADDLIB("rexxsupport.library",0,-30,0) → 1
CALL ADDLIB "EtherNet",-20       /*A gateway*/
```

ADDRESS() ADDRESS()

Returns the current host address string. The host address is the message port to which commands will be sent. The SHOW() function can be used to check whether the required external host is actually available. See also SHOW(). For example:

```
SAY ADDRESS()    → REXX
```

ARG() ARG([number][,'EXISTS'|'OMITTED'])

ARG() returns the number of arguments supplied to the current environment. If only the number parameter is supplied, the corresponding argument string is returned. If a number and the Exists or Omitted keyword is given, the Boolean return indicates the status of the corresponding argument. The existence or omission test does not indicate whether the string has a null value, but only whether a string was supplied. For example:

```
/*Assume arguments were: ('one',,10)*/
SAY ARG()    → 3
SAY ARG(1)    → one
SAY ARG(2,'O')    → 1
```

B2C() B2C(string)

Converts a string of binary digits (0,1) into the corresponding (packed) character representation. The conversion is the same as though the argument string had been specified as a literal binary string (e.g. '1010'B). Blanks are permitted in the string, but only at byte boundaries. This function is particularly useful for creating

strings that are to be used as bit masks. See also X2C(). For example:

```
SAY B2C('00110011')    → 3
SAY B2C('01100001')    → a
```

BITAND() BITAND(string1,string2[,pad])

The argument strings are logically ANDed together, with the length of the result being the longer of the two operand strings. If a pad character is supplied, the shorter string is padded on the right. Otherwise, the operation terminates at the end of the shorter string, and the remainder of the longer string is appended to the result. For example:

```
BITAND('0313'x,'FFF0'x)    → '0310'x
```

BITCHG() BITCHG(string,bit)

Changes the state of the specified bit in the argument string. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string. For example:

```
BITCHG('0313'x,4)    → '0303'x
```

BITCLR() BITCLR(string,bit)

Clears (sets to zero) the specified bit in the argument string. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string. For example:

```
BITCLR('0313'x,4)    → '0303'x
```

BITCOMP() BITCOMP(string1,string2[,pad])

Compares the argument strings bit-by-bit, starting at bit number 0. The returned value is the bit number of the first bit in which the strings differ, or –1 if the strings are identical. For example:

```
BITCOMP('7F'x,'FF'x)    → 7    /*Seventh bit*/
BITCOMP('FF'x,'FF'x)    → -1
```

BITOR() BITOR(string1,string2[,pad])

The argument strings are logically ORed together, with the length of the result being the longer of the two operand strings. If a pad character is supplied, the shorter string is padded on the right. Otherwise, the operation terminates at the end of the shorter string, and the remainder of the longer string is appended to the result. For example:

```
BITOR('0313'x,'003F'x)    → '033F'x
```

BITSET() BITSET(string,bit)

Sets the specified bit in the argument string to 1. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string. For example:

```
BITSET('0313'x,2)    → '0317'x
```

BITTST() BITTST(string,bit)

The Boolean return indicates the state of the specified bit in the argument string. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string. For example:

```
BITTST('0313'x,4)    → 1
```

BITXOR() BITXOR(string1,string2[,pad])

The argument strings are logically and exclusively-ORed together, with the length of the result being the longer of the two operand strings. If a pad character is supplied, the shorter string is padded on the right. Otherwise, the operation terminates at the end of the shorter string, and the remainder of the longer string is appended to the result. For example:

```
BITXOR('0313'x,'001F'x)    → '030C'X
```

C2B() C2B(string)

Converts the character string into the equivalent string of binary digits. See also C2X(). For example:

```
SAY C2B('abc')    → 011000010110001001100011
```

C2D() C2D(string[,n])

Converts the string argument from its character representation to the corresponding decimal number, expressed as ASCII digits (0-9). If n is supplied, the character string is considered to be a number expressed in n bytes. The string is truncated or padded with nulls on the left as required, and the sign bit is extended for the conversion. For example:

```
SAY C2D('0020'x)    → 32
SAY C2D('FFFF ffff'x)    → -1
SAY C2D('FF0100'x,2)    → 256
```

C2X() C2X(string)

Converts the string argument from its character representation to the corresponding hexadecimal number, expressed as the ASCII characters 0-9 and A-F. See also C2B(). For example:

```
SAY C2X('abc')    → 616263
```

CENTER() CENTER(string,length[,pad])
CENTRE() CENTRE(string,length[,pad])

Centers the string argument in a string with the specified length. If the length is longer than that of the string, pad characters or blanks are added as necessary. For example:

```
SAY CENTER('abc',6)        → ' abc  '
SAY CENTER('abc',6,'+')    → '+abc++'
SAY CENTER('123456',3)     → '234'
```

CLOSE() CLOSE(file)

Closes the file specified by the given logical name. The returned value is a Boolean success flag and will be 1 unless the specified file was not open. For example:

```
SAY CLOSE('input')    → 1
```

COMPARE() COMPARE(string1,string2[,pad])

Compares two strings and returns the index of the first position in which they differ or 0 if the strings are identical. The shorter string is padded as required using the supplied character or blanks. For example:

```
SAY COMPARE('abcde','abcce')    → 4
SAY COMPARE('abcde','abcde')    → 0
SAY COMPARE('abc++','abc+-','+')    → 5
```

COMPRESS() COMPRESS(string[,list])

If the list argument is omitted, the function removes leading, trailing or embedded blank characters from the string argument. If the optional list is supplied, it specifies the characters to be removed from the string. For example:

```
SAY COMPRESS('  why not  ')    → whynot
SAY COMPRESS('++12-34-+','+-')    → 1234
```

COPIES() COPIES(string,number)

Creates a new string by concatenating the specified number of copies of the original. The number argument may be zero, in which case the null string is returned. For example:

```
SAY COPIES('abc',3)    → abcabcabc
```

D2C() D2C(number)

Creates a string whose value is the binary (packed) representation of the given decimal number. For example:

```
D2C(65)    → A
```

D2X() D2X(number[,digits])

Converts a decimal number to hexadecimal. For example:

```
D2X(31)    → 1F
```

DATATYPE() DATATYPE(string[,option])

If only the string argument is specified, DATATYPE() tests whether the string parameter is a valid number and returns either NUM or CHAR. If an option keyword is given, the Boolean return indicates whether the string satisfied the requested test. The following option keywords are recognized:

ALPHANUMERIC	Accepts alphabetic (A-Z, a-z) or numeric (0-9) characters
BINARY	Accepts a binary digits string
LOWERCASE	Accepts lowercase alphabetic (a-z) characters
MIXED	Accepts mixed upper/lowercase characters

NUMERIC	Accepts valid numbers
SYMBOL	Accepts valid REXX symbols
UPPER	Accepts uppercase alphabetic (A-Z) characters
WHOLE	Accepts integer numbers
X	Accepts Hex digits strings

For example:

```
SAY DATATYPE('123')    → NUM
SAY DATATYPE('1a f2','X')    → 1
SAY DATATYPE('aBcde','L')    → 0
```

DATE() DATE([option][,date][,format])

Returns the current date in the specified format. The default ('NORMAL') option returns the date in the form DD MMM YYYY, as in 20 APR 1988. The options recognized are:

BASEDATE	The number of days since January 1, 0001
CENTURY	The number of days since January 1 of the century
DAYS	The number of days since January 1 of the current year
EUROPEAN	The date in the form DD/MM/YY
INTERNAL	Internal system days
JULIAN	The date in the form YYDDD
MONTH	The current month (in mixed case)
NORMAL	The date in the form DD MMM YYYY
ORDERED	The date in the form YY/MM/DD
SORTED	The date in the form YYYYMMDD
USA	The date in the form MM/DD/YY
WEEKDAY	The day of the week (in mixed case)

These options can be shortened to just the first character.

The DATE() function also accepts optional second and third arguments to supply the date either in the form of internal system days or in the 'sorted' form YYYYMMDD. The second argument is specifying either system days (the default) or a sorted date. The

third argument specifies the form of the date and can be either 'I' or 'S'. The current date in system days can be retrieved using DATE('INTERNAL'). For example:

```
SAY DATE()              → 14 Jul 1992
SAY DATE('M')           → July
SAY DATE(S)             → 19920714
SAY DATE('S',DATE('I')+21)    → 19920804
SAY DATE('W',19890609,'S')    → Friday
```

DELSTR() DELSTR(string,n[,length])

Deletes the substring of the string argument beginning with the nth character for the specified length in characters. The default length is the remaining length of the string. For example:

```
SAY DELSTR('123456',2,3)    → 156
```

DELWORD() DELWORD(string,n[,length])

Deletes the substring of the string argument beginning with the nth word for the specified length in words. The default length is the remaining length of the string. The deleted string includes any trailing blanks following the last word. For example:

```
SAY DELWORD('Tell me a story',2,2)    → 'Tell story'
SAY DELWORD('one two three',3)    → 'one two '
```

DIGITS() DIGITS()

Returns the current Numeric Digits setting. For example:

```
NUMERIC DIGITS 6
SAY DIGITS()     → 6
```

EOF() EOF(file)

Checks the specified logical file name and returns the Boolean value 1 (True), if the end-of-file has been reached, and 0 (False) otherwise. For example:

```
SAY EOF(infile)    → 1
```

ERRORTEXT() ERRORTEXT(n)

Returns the error message associated with the specified ARexx error code. The null string is returned if the number is not a valid error code. For example:

```
SAY ERRORTEXT(41)    → Invalid expression
```

EXISTS() EXISTS(filename)

Tests whether an external file of the given filename exists. The name string may include device and directory specifications. For example:

```
SAY EXISTS('SYS:C/ED')    → 1
```

EXPORT() EXPORT(address[,string][,length][,pad])

Copies data from the optional string into a previously-allocated memory area. This memory area must be specified as a four-byte address. The length parameter specifies the maximum number of characters to be copied. The default is the length of the string. If the specified length is longer than the string, the remaining area is filled with the pad character or nulls ('00'x). The returned value is the number of characters copied.

Caution **Any area of memory can be overwritten, possibly causing a system crash. Task switching is forbidden while the copy is being done, so system performance may be degraded if long strings are copied.**

See also IMPORT() and STORAGE(). For example:

```
count = EXPORT('0004 0000'x,'The answer')
```

FIND() FIND(string,phrase)

The FIND() function locates a phrase of words in a larger string of words and returns the word number of the matched position. For example:

```
SAY FIND('Now is the time','is the')    → 2
```

FORM() FORM()

Returns the current NUMERIC FORM setting. For example:

```
NUMERIC FORM SCIENTIFIC
SAY FORM()    → SCIENTIFIC
```

FREESPACE() FREESPACE(address,length)

Returns a block of memory of a given length to the interpreter's internal pool. The address argument must be a 4 byte string obtained by a prior call to GETSPACE(), the internal allocator. It is not always necessary to release internally allocated memory, since it will be released to the system when the program terminates. However, if a very large block has been allocated, returning it to the pool may avoid memory space problems. The return value is a boolean success flag. See also GETSPACE().

Calling FREESPACE() with no arguments will return the amount of memory available in the interpreter's internal pool. For example:

```
FREESPACE('00042000'x,32)    → 1
```

FUZZ() FUZZ()

Returns the current NUMERIC FUZZ setting. For example:

```
NUMERIC FUZZ 3
SAY FUZZ()    → 3
```

GETCLIP() GETCLIP(name)

Searches the Clip List for an entry matching the supplied name parameter and returns the associated value string. The name matching is case-sensitive. The null string is returned if the name cannot be found. See also SETCLIP(). For example:

```
/*Assume 'numbers' contains 'PI=3.14159'*/
SAY GETCLIP('numbers')    → PI=3.14159
```

GETSPACE() GETSPACE(length)

Allocates a block of memory of the specified length from the interpreter's internal pool. The returned value is the four-byte address of the allocated block, which is not cleared or otherwise initialized. Internal memory is automatically returned to the system when the ARexx program terminates, so this function should not be used to allocate memory for use by external programs. The REXXSupport.Library includes the function ALLOCMEM(), which allocates memory from the system free list. See also FREESPACE(). For example:

```
SAY C2X(GETSPACE(32))    → '0003BF40'x
```

HASH() HASH(string)

Returns the hash attribute of a string as a decimal number and updates the internal hash value of the string. For example:

```
SAY HASH('1')    → 49
```

IMPORT() IMPORT(address[,length])

Creates a string by copying data from the specified four-byte address. If the length parameter is not supplied, the copy terminates when a null byte is found. See also EXPORT(). For example:

```
extval = IMPORT('0004 0000'x,8)
```

INDEX() INDEX(string,pattern[,start])

Searches for the first occurrence of the pattern argument in the string argument, beginning at the specified start position. The default start position is 1. The returned value is the index of the matched pattern or 0 if the pattern was not found. For example:

```
SAY INDEX("123456","23")    → 2
SAY INDEX("123456","77")    → 0
SAY INDEX("123123","23",3)  → 5
```

INSERT() INSERT(new,old[,start][,length][,pad])

Inserts the new string into the old string after the specified start position. The default starting position is 0. The new string is truncated or padded to the specified length as required, using the supplied pad character or blanks. If the start position is beyond the end of the old string, the old string is padded on the right. For example:

```
SAY INSERT('ab','12345')    → ab12345
SAY INSERT('123','++',3,5,'-')    → ++-123--
```

LASTPOS() LASTPOS(pattern,string[,start])

Searches backwards for the first occurrence of the pattern argument in the string argument, beginning at the specified start position. The default starting position is the end of the string. The returned value is the index of the matched pattern or 0 if the pattern was not found. For example:

```
SAY LASTPOS('2','1234')    → 2
SAY LASTPOS('2','1234234')    → 5
SAY LASTPOS('2','123234',3)    → 2
SAY LASTPOS('2','13579')    → 0
```

LEFT() LEFT(string,length[,pad])

Returns the leftmost substring in the given string argument with the specified length. If the substring is shorter than the requested length, it is padded on the right with the supplied pad character or blanks. For example:

```
SAY LEFT('123456',3)    → 123
SAY LEFT('123456',8,'+')    → 123456++
```

LENGTH() LENGTH(string)

Returns the length of the string. For example:

```
SAY LENGTH('three')    → 5
```

LINES() LINES(file)

Returns the number of lines queued or typed ahead at the logical file, which must refer to an interactive stream. The line count is obtained as the secondary result of a WaitForChar() call. For example:

```
PUSH 'a line'
PUSH 'another one'
SAY LINES(STDIN)    → 2
```

MAX() MAX(number,number[,number, . . .])

Returns the maximum of the supplied arguments, all of which must be numeric. At least two parameters must be supplied. For example:

```
SAY MAX(2.1,3,-1)    → 3
```

MIN() MIN(number,number[,number, . . .])

Returns the minimum of the supplied arguments, all of which must be numeric. At least two parameters must be supplied. For example:

```
SAY MIN(2.1,3,-1)    → -1
```

OPEN() OPEN(file,filename[,'APPEND' | 'READ' | 'WRITE'])

Opens an external file for the specified operation. The file argument defines the logical name by which the file will be referenced. The filename is the external name of the file and may include device and directory specifications. The function returns a Boolean value that indicates whether the operation was successful. There is no limit to the number of files that can be open simultaneously, and all open files are closed automatically when the program exits. See also CLOSE(), READ(), and WRITE(). For example:

```
SAY OPEN('MyCon','CON:160/50/320/100/MyCon/cds') → 1
SAY OPEN('outfile','ram:temp','W')    → 1
```

OVERLAY() OVERLAY(new,old[,start][,length][,pad])

Overlays the new string onto the old string beginning at the specified start position, which must be positive. The default starting position is 1. The new string is truncated or padded to the specified length as required, using the supplied pad character or blanks. If the start position is beyond the end of the old string, the old string is padded on the right. For example:

```
SAY OVERLAY('bb','abcd')    → bbcd
SAY OVERLAY('4','123',5,5,'-')  → 123-4----
```

POS() POS(pattern,string[,start])

Searches for the first occurrence of the pattern argument in the string argument, beginning at the position specified by the start argument. The default starting position is 1. The value is the index of the matched string or 0 if the pattern wasn't found. For example:

```
SAY POS('23','123234')        → 2
SAY POS('77','123234')        → 0
SAY POS('23','123234',3)      → 4
```

PRAGMA() PRAGMA(option[,value])

This function allows a program to change various attributes relating to the system environment within which the program executes. The option argument is a keyword that specifies an environmental attribute. The value argument supplies the new attribute value to be installed. The value returned by the function depends on the attribute selected. Some attributes return the previous value installed, while others may simply set a Boolean success flag.

The currently defined option keywords are:

DIRECTORY Specifies a new current directory. The current directory is used as the root for filenames that do not explicitly include a device specification. The return is the old directory name. PRAGMA('D') is equivalent to PRAGMA('D',''). It returns the path of the current directory without changing the directory.

PRIORITY Specifies a new task priority. The priority value must be an integer in the range –128 to 127, but the practical range is much more limited. ARexx programs should never be run at a priority higher than that of the resident process, which currently runs at a priority of 4. The returned value is the previous priority level.

ID Returns the task ID (the address of the task block) as an 8-character hex string. The task ID is a unique identifier of the particular ARexx invocation and may be used to create a unique name for it.

STACK Specifies a new stack value for your current ARexx program. When a new stack value is declared, the previous stack value is returned.

The currently implemented options are:

PRAGMA('W',{'NULL'|'WORKBENCH'}) Controls the task's WindowPtr field. Setting it to 'NULL' will suppress any requesters that might otherwise be generated by a DOS call.

PRAGMA('*'[,name]) Defines the specified logical name as the current ("*") console handler, allowing the user to open two streams on one window. If the name is omitted, the console handler is set to that of the client's process.

```
SAY PRAGMA('D','DF0:C')    → Extras
SAY PRAGMA('D','DF1:C')    → Workbench:C
SAY PRAGMA('PRIORITY',-5)    → 0
SAY PRAGMA('ID')    → 00221ABC
CALL PRAGMA '*',STDOUT
SAY PRAGMA("STACK",8092)    → 4000
```

RANDOM() RANDOM([MIN][,MAX][,seed])

Returns a pseudo-random integer in the interval specified by the min and max arguments. The default minimum value is 0 and the default maximum value is 999. The interval max-min must be less than or equal to 1000. If a greater range of random integers is required, the values from the RANDU() function can be suitably scaled and translated. The seed argument can be supplied to initialize the internal state of the random number generator. See also RANDU(). For example:

```
thisroll = RANDOM(1,6)    /*Might be 1*/
nextroll = RANDOM(1,6) /*Might be snake eyes*/
```

RANDU() RANDU([seed])

Returns a uniformly-distributed, pseudo-random number between 0 and 1. The number of digits of precision in the result is always equal to the current Numeric Digits setting. With the choice of suitable scaling and translation values, RANDU() can be used to generate pseudo-random numbers on an arbitrary interval.

The optional integer seed argument is used to initialize the internal state of the random number generator. See also RANDOM(). For example:

```
firsttry = RANDU()    /*0.371902021?*/
NUMERIC DIGITS 3
tryagain = RANDU()    /*0.873?*/
```

READCH() READCH(file,length)

Reads the specified number of characters from the given logical file into a string. The length of the returned string is the actual number of characters read and may be less than the requested length if, for example, the end-of-file was reached. See also READLN(). For example:

```
instring = READCH('input',10)
```

READLN() READLN(file)

Reads characters from the given logical file into a string until a newline character is found. The returned string does not include the newline. See also READCH(). For example:

```
instring = READLN('MyFile')
```

REMLIB() REMLIB(name)

Removes an entry with the given name from the Library List maintained by the resident process. The Boolean return is 1 if the entry was found and successfully removed. This function does not make a distinction between function libraries and function hosts,

but simply removes a named entry. See also ADDLIB(). For example:

```
SAY REMLIB('MyLibrary.library')    → 1
```

REVERSE() REVERSE(string)

Reverses the sequence of characters in the string. For example:

```
SAY REVERSE('?ton yhw')  → why not?
```

RIGHT() RIGHT(string,length[,pad])

Returns the rightmost substring in the given string argument with the specified length. If the substring is shorter than the requested length, it is padded on the left with the supplied pad character or blanks. For example:

```
SAY RIGHT('123456',4)    → 3456
SAY RIGHT('123456',8,'+')    → ++123456
```

SEEK() SEEK(file,offset[,'BEGIN'|'CURRENT'|'END'])

Moves to a new position in the given logical file, specified as an offset from an anchor position. The default anchor is Current. The returned value is the new position relative to the start of the file. For example:

```
SAY SEEK('input',10,'B')    → 10
SAY SEEK('input',0,'E') → 356 /*file length*/
```

SETCLIP() SETCLIP(name[,value])

Adds a name-value pair to the Clip List maintained by the resident process. If an entry of the same name already exists, its value is updated to the supplied value string. Entries may be removed by specifying a null value. The function returns a Boolean value that indicates whether the operation was successful. For example:

```
SAY SETCLIP('path','DF0:s')    → 1
SAY SETCLIP('path')    → 1
```

SHOW() SHOW(option[,name][,pad])

Returns the names in the resource list specified by the option argument, or tests to see whether an entry with the specified name is available. The currently implemented options keywords are:

CLIP	Examines the names in the Clip List
FILES	Examines the currently open logical file names
LIBRARIES	Examines the names in the Library List, which are either function libraries or function hosts.
PORTS	Examines the names in the system Ports List

If the name argument is omitted, the function returns a string with the resource names separated by a blank space or the pad character, if one was supplied. If the name argument is given, the returned Boolean value indicates whether the name was found in the resource list. The name entries are case-sensitive.

SIGN() SIGN(number)

Returns 1 if the number argument is positive or zero and –1 if the number is negative. The argument must be numeric. For example:

```
SAY SIGN(12)    → 1
SAY SIGN(-33)    → -1
```

SOURCELINE() SOURCELINE([line])

Returns the text for the specified line of the currently executing ARexx program. If the line argument is omitted, the function returns the total number of lines in the file. This function is often used to embed "help" information in a program. For example:

```
/*A simple test program*/
SAY SOURCELINE()    → 3
SAY SOURCELINE(1) → /*A simple test program*/
```

SPACE() SPACE(string,n[,pad])

Reformats the string argument so that there are n spaces (blank characters) between each pair of words. If the pad character is specified, it is used instead of blanks as the separator character. Specifying n as 0 will remove all blanks from the string. For example:

```
SAY SPACE('Now is the time',3)
    → 'Now   is   the   time'
SAY SPACE('Now is the time',0)
    → 'Nowisthetime'
SAY SPACE('1 2 3',1,'+')    → '1+2+3'
```

STORAGE() STORAGE([address][,string][,length][,pad])

STORAGE() with no arguments returns the available system memory. If the address argument is given, it must be a four-byte string. The function copies data from the (optional) string to the indicated memory address. The length parameter specifies the maximum number of bytes to be copied and defaults to the length of the string. If the specified length is longer than the string, the remaining area is filled with the pad character or nulls ('00'x).

The returned value is the previous contents of the memory area. This can be used in a subsequent call to restore the original contents. See also EXPORT().

Caution **Any area of memory can be overwritten, possibly causing a system crash. Task switching is forbidden while the copy is being done, so system performance may be degraded if long strings are copied.**

For example:

```
SAY STORAGE()    → ' 248400
oldval = STORAGE('0004 0000'x,'The answer')
CALL STORAGE '0004 0000'x,,32,'+'
```

STRIP() STRIP(string[,{'B' | 'L' | 'T'}][,pad])

If neither of the optional parameters is supplied, the function removes both leading and trailing blanks from the string argument. The second argument specifies whether Leading, Trailing or Both (leading and trailing) characters are to be removed. The optional pad (or unpad) argument selects the character to be removed. For example:

```
SAY STRIP('  say what?  ')    → 'say what?'
SAY STRIP('  say what?  ','L') → 'say what?
SAY STRIP('++123+++','B','+')   → '123'
```

SUBSTR() SUBSTR(string,start[,length][,pad])

Returns the substring of the string argument beginning at the specified start position for the specified length. The starting position must be positive, and the default length is the remaining length of the string. If the substring is shorter than the requested length, it is padded on the right with the blanks or the specified pad character. For example:

```
SAY SUBSTR('123456',4,2)    → 45
SAY SUBSTR('myname',3,6,'=')    → name==
```

SUBWORD() SUBWORD(string,n[,length])

Returns the substring of the string argument beginning with the nth word for the specified length in words. The default length is the remaining length of the string. The returned string will never have leading or trailing blanks. For example:

```
SAY SUBWORD('Now is the time  ',2,2)  → is the
```

SYMBOL() SYMBOL(name)

Tests whether the name argument is a valid ARexx symbol. If the name is not a valid symbol, the function returns the string BAD. If the symbol is uninitialized, the returned string is LIT. If the symbol has been assigned a value, VAR is returned. For example:

```
SAY SYMBOL('J')    → VAR
SAY SYMBOL('x')    → LIT
SAY SYMBOL('++')    → BAD
```

TIME() TIME(option)

Returns the current system time or controls the internal elapsed time counter. The valid option keywords are:

CIVIL	Current time in 12 hour format (a.m./p.m.) hours/minutes
ELAPSED	Elapsed time in seconds since program start
HOURS	Current time in hours since midnight
MINUTES	Current time in minutes since midnight
NORMAL	Current time in 24 hour format (hours/minutes/seconds)
RESET	Reset the elapsed time clock
SECONDS	Current time in seconds since midnight

If no option is specified, the function returns the current system time in the form HH:MM:SS. For example:

```
/*Suppose that the time is 1:02 AM . . .*/
SAY TIME('C')      → 1:02 AM
SAY TIME('HOURS')     → 1
SAY TIME('M')      → 62
SAY TIME('N')      → 01:02:54
SAY TIME('S')      → 3720
call TIME('R')     /*reset timer*/
SAY TIME('E')      → .020
SAY TIME()         → 01:02:00
```

TRACE() TRACE(option)

Sets the tracing mode (see Chapter 6) to that specified by the option keyword, which must be one of the valid alphabetic or prefix options. The TRACE() function will alter the tracing mode even during interactive tracing, when TRACE instructions in the source program are ignored. The returned value is the mode in effect before the function call. This allows the previous trace mode to be restored later. For example:

```
/*Assume tracing mode is ?ALL*/
SAY TRACE('Results')    → ?A
```

TRANSLATE() TRANSLATE(string[,output][,input][,pad])

This function constructs a translation table and uses it to replace selected characters in the argument string. If only the string argument is given, it is translated to uppercase. If an input table is supplied, it modifies the translation table so that characters in the argument string that occur in the input table are replaced with the corresponding character in the output table. Characters beyond the end of the output table are replaced with the specified pad character or a blank. The result string is always of the same length as the original string. The input and output tables may be of any length. For example:

```
SAY TRANSLATE("abcde","123","cbade","+") → 321++
SAY TRANSLATE("low")    → LOW
SAY TRANSLATE("0110","10","01")    → 1001
```

TRIM() TRIM(string)

Removes trailing blanks from the string argument. For example:

```
SAY length(TRIM(' abc    ')    → 4
```

TRUNC() TRUNC(number[,places])

Returns the integer part of the number argument followed by the specified number of decimal places. The default number of decimal places is 0. The number is padded with zeroes as necessary. For example:

```
SAY TRUNC(123.456)    → 123
SAY TRUNC(123.456,4)    → 123.4560
```

UPPER() UPPER(string)

Translates the string to uppercase. The action of this function is equivalent to that of TRANSLATE(string), but it is slightly faster for short strings. For example:

```
SAY UPPER('One Fine Day')    →  ONE FINE DAY
```

VALUE() VALUE(name)

Returns the value of the symbol represented by the name argument. For example:

```
/*Assume that J has the value 12*/
SAY VALUE('j')    → 12
```

VERIFY() VERIFY(string,list[,'MATCH'])

Returns the index of the first character in the string argument which is not contained in the list argument or 0 if all of the characters are in the list. If the MATCH keyword is supplied, the function returns the index of the first character which is in the list or 0 if none of the characters are in the list. For example:

```
SAY VERIFY('123456','0123456789')    → 0
SAY VERIFY('123a56','0123456789')    → 4
SAY VERIFY('123a45','abcdefghij','m')    → 4
```

WORD() WORD(string,n)

Returns the nth word in the string argument or the null string if there are fewer than n words. For example:

```
SAY WORD('Now is the time  ',2)   → is
```

WORDINDEX() WORDINDEX(string,n)

Returns the starting position of the nth word in the argument string or 0 if there are fewer than n words. For example:

```
SAY WORDINDEX('Now is the time  ',3)   → 8
```

WORDLENGTH() WORDLENGTH(string,n)

Returns the length of the nth word in the string argument. For example:

```
SAY WORDLENGTH('one two three',3)   → 5
```

WORDS() WORDS(string)

Returns the number of words in the string argument. For example:

```
SAY WORDS("You don''t SAY!")   → 3
```

WRITECH() WRITECH(file,string)

Writes the string argument to the given logical file. The returned value is the actual number of characters written. For example:

```
SAY WRITECH('output','Testing')   → 7
```

WRITELN() WRITELN(file,string)

Writes the string argument to the given logical file with a newline appended. The returned value is the actual number of characters written. For example:

```
SAY WRITELN('output','Testing')    → 8
```

X2C() X2C(string)

Converts a string of hex digits into the (packed) character representation. Blank characters are permitted in the argument string at byte boundaries. For example:

```
SAY X2C('12ab')    →'12ab'x
SAY X2C('12 ab')    → '12ab'x
SAY X2C(61)    → a
```

X2D() X2D(hex,digits)

Converts a hexadecimal number to decimal. For example:

```
SAY X2D('1f')    → 31
```

XRANGE() XRANGE([start][,end])

Generates a string consisting of all characters numerically between the specified start and end values. The default start character is '00'x, and the default end character is 'FF'x. Only the first character of the start and end arguments is significant. For example:

```
SAY XRANGE()    → '00010203  . . .  FDFEFF'x
SAY XRANGE('a','f')    → 'abcdef'
SAY XRANGE(,'0A'x)→ '000102030405060708090A'x
```

Example Program

The following example program illustrates many of the built-in functions that manipulate character strings.

Program 13. Changestrings.rexx

```
/*This ARexx program shows the effect of built-in
functions that change strings. The functions come in
two groups: one that manipulates individual
characters and one that manipulates whole strings.*/

teststring1 = " every  good   boy   does   fine  "

/*The first group is composed of the functions
STRIP(), COMPRESS(), SPACE(), TRIM(), TRANSLATE(),
DELSTR(), DELWORD(), INSERT(), OVERLAY(), and
REVERSE().*/

/*STRIP() removes only leading and trailing characters.*/

/*Print the original string, for comparison. We put a
period at the end of the string, so you can see what
happens to the spaces at the end of the string.*/
SAY " every  good   boy   does   fine   "

/*The same string stripped of leading and trailing
spaces*/
SAY STRIP(" every  good   boy   does   fine   ")"."

/*Failed attempt to remove leading and trailing "e"s*/
SAY STRIP(" every  good   boy   does   fine
",,"e")"."

/*The "e"'s were protected by the leading and
trailing spaces.  Removing them exposes the "e"'s to
the effects of STRIP()*/
SAY STRIP("every  good   boy   does   fine",,"e")"."

/*Remove "e"'s and spaces from the original string*/
SAY STRIP(" every  good   boy   does   fine   ",," e")"."

/*We are now using the variable "teststring1",
defined above. Remove only the trailing spaces in the
test string.*/
SAY STRIP(teststring1, T)"."
```

```
/*Remove the trailing spaces and the "e"*/
SAY STRIP(teststring1,T," e")"."
.
/*Compress() removes characters anywhere in the
string.  This removes all blanks from the test string*/
SAY COMPRESS(teststring1)

CALL TIME('r')
SAY TIME('Civil') /*Civilian time HH:MM{AM | PM}*/
SAY TIME('h')     /*Hours since midnight*/
SAY TIME('m')     /*Minutes since midnight*/
SAY TIME('s')     /*Seconds since midnight*/
SAY TIME('e')     /*Elapsed time since program start*/

/*Function: TRACE  Usage: TRACE([option])*/
SAY TRACE()
SAY TRACE(TRACE())   /*Leave it unchanged*/
/*Function:TRANSLATE Usage: TRANSLATE(string[,output]
[,input][,pad])*/
SAY TRANSLATE('aBCdef')   /*Translate to Uppercase*/
SAY TRANSLATE ('abcdef','1234')
SAY TRANSLATE('654321','abcdef','123456')
SAY TRANSLATE('abcdef','123','abcdef','+')

/*Function: TRIM  Usage: TRIM(string)*/
SAY TRIM('   abc  ')

/*Function: TRUNC  Usage: TRUNC(number[,places])*/
SAY TRUNC(123.456)
SAY '$'TRUNC(134566.123,2)

/*Function: UPPER  Usage: UPPER(string)*/
SAY UPPER('aBCdef12')

/*Function: VALUE  Usage: VALUE(name)*/
abc = 'my name'
SAY VALUE('abc')

/*Function: VERIFY  Usage: VERIFY(string,list[,'M'])*/
SAY VERIFY('123a45','0123456789')
SAY VERIFY('abc3de','012456789','M')

/*Function: WORD  Usage: WORD(string,n)*/
SAY WORD('Now is the time',3)

/*Function: WORDINDEX  Usage: WORDINDEX(string,n)*/
SAY WORDINDEX('Now is the time  ',3)
```

```
/*Function: WORDLENGTH  Usage: WORDLENGTH(string,n)*/
SAY WORDLENGTH('Now is the time  ',4)

/*Function: WORDS  Usage: WORDS(string)*/
SAY WORDS('Now is the time')
/*Function: WRITECH  Usage: WRITECH(logical,string)*/
IF OPEN('test','ram:test$$','W') THEN DO
   SAY WRITECH('test','message') /*Write the string*/
   CALL CLOSE 'test'
END

/*Function: WRITELN  Usage: WRITELN(logical,string)*/
IF OPEN('test','ram:test$$','W') THEN DO
   SAY WRITELN('test','message')
   /*Write the string (with newline)*/
   CALL CLOSE 'test'
END

/*Function: X2C  Usage: X2C(hexstring)*/
SAY X2C('616263')   /*Convert to character (pack)*/

/*Function: XRANGE  Usage: XRANGE([start][,end])*/
SAY C2X(xrange('f0'x))
SAY XRANGE('a','g')
EXIT
```

The output of Program 13 is:

```
 every  good   boy   does   fine
every  good   boy   does   fine.
 every  good   boy   does   fine .
very  good   boy   does   fin.
very  good   boy   does   fin.
 every  good   boy   does   fine.
 every  good   boy   does   fin.
everygoodboydoesfine
1:23PM   /*These results vary depending*/
13       /*on the time the program is run.*/
803
48199
0.80
N
N
ABCDEF
abcdef
fedcba
123+++
   abc
123
$134566.12
ABCDEF12
my name
4
0
the
8
4
4
7
8
abc
F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF
abcdefg
```

REXXSupport.Library Functions

The functions listed in this section are part of the REXXSupport.library. They may only be used if this library has been opened. Below is an example that shows you how to open this library.

Program 14. OpenLibrary.rexx

```
/*Add rexxsupport.library if it isn't already open.*/
IF ~ SHOW('L', "rexxsupport.library") THEN DO
/*If the library isn't open, try to open it*/
IF ADDLIB('rexxsupport.library', 0, -30,0)
THEN SAY "Added rexxsupport.library."
ELSE DO
   SAY 'ARexx support library not available, exiting'
   EXIT 10     /*Exit if ADDLIB() failed*/
   END
END
```

ALLOCMEM() ALLOCMEM(length[,attribute])

Allocates a block of memory of the specified length from the system free-memory pool and returns its address as a four-byte string. The optional attribute parameter must be a standard EXEC memory allocation flag, supplied as a four-byte string. The default attribute is for "PUBLIC" memory (not cleared). Refer to the *Amiga ROM Kernel Reference Manual: Libraries* for information on memory types and attribute parameters.

This function should be used whenever memory is allocated for use by external programs. It is the user's responsibility to release the memory space when it is no longer needed. See also FREEMEM(). For example:

```
SAY C2X(ALLOCMEM(1000))    → 00050000
SAY C2X(ALLOCMEM (1000,'00 01 00 0 1'X )) → 00228400
/*1000 bytes of CLEAR Public memory*/
```

CLOSEPORT() CLOSEPORT(name)

Closes the message port specified by the name argument, which must have been allocated by a call to OPENPORT() within the current ARexx program. Any messages received but not yet REPLYed are automatically returned with the return code set to 10. See also OPENPORT(). For example:

```
CALL CLOSEPORT myport
```

FREEMEM() FREEMEM(address,length)

Releases a block of memory of the given length to the system free list. The address parameter is a four-byte string, typically obtained by a prior call to ALLOCMEM(). FREEMEM() cannot be used to release memory allocated using GETSPACE(), the ARexx internal memory allocator. The returned value is a Boolean success flag. See also ALLOCMEM(). For example:

```
MemoryRequest = 1024
MyMem = ALLOCMEM(MemoryRequest)
SAY C2X(MyMem)                  → 07C987B0
SAY FREEMEM(MyMem, MemoryRequest)    → 1
/*Or: SAY FREEMEM('07C987B0'x,1024)*/
```

Caution **Before your program terminates, you must use a matching FREEMEM() to release the exact amount of memory you allocated with each ALLOCMEM(). Otherwise, you may crash the system or leave memory unavailable until you reboot.**

GETARG() GETARG(packet[,n])

Extracts a command, function name, or argument string from a message packet. The packet argument must be a four-byte address obtained from a prior call to GETPKT(). The optional [n] argument specifies the slot containing the string to be extracted and must be less than or equal to the actual argument count for the packet. Commands and function names are always in slot 0. Function packets may have argument strings in slots 1-15. For example:

```
command = GETARG(packet)
function = GETARG(packet,0)   /*name string*/
arg1 = GETARG(packet,1)       /*1st argument*/
```

GETPKT() GETPKT(name)

Checks the message port specified by the name argument to see whether any messages are available. The named message port must have been opened by a prior call to OPENPORT() within the

current ARexx program. The returned value is the four-byte address of the first message packet, or '0000 0000'x if no packets were available.

The function returns immediately whether or not a packet is enqueued at the message port. Programs should never be designed to "busy-loop" on a message port. If there is no useful work to be done until the next message packet arrives, the program should call WAITPKT() and allow other tasks to proceed. See also WAITPKT(). For example:

```
packet = GETPKT('MyPort')
```

OPENPORT() OPENPORT(name)

Creates a public message port with the given name. The returned Boolean value indicates whether the port was successfully opened. An initialization failure will occur if another port of the same name already exists or if a signal bit couldn't be allocated. The message port is allocated as a Port Resource node and is linked into the program's global data structure. Ports are automatically closed when the program exits and any pending messages are returned to the sender. See also CLOSEPORT(). For example:

```
success = OPENPORT("MyPort")
```

REPLY() REPLY(packet,rc)

Returns a message packet to the sender, with the primary result field set to the value given by the rc argument. The secondary result is cleared. The packet argument must be supplied as a four-byte address, and the rc argument must be a whole number. For example:

```
CALL REPLY(packet,10) /*Error return*/
```

SHOWDIR() SHOWDIR(directory[,'ALL' | 'FILE' | 'DIR'][,pad])

Returns the contents of the specified directory as a string of names separated by blanks. The second parameter is an option keyword

that selects whether all entries, only files, or only subdirectories, will be included. For example:

```
SAY SHOWDIR('SYS:REXXC', 'f', ';')
→ WaitForPort;TS;TE;TCO;RXSET;RXLIB;RXC;RX;HI
```

SHOWLIST() SHOWLIST({'A' | 'D' | 'H' | 'I' | 'L' | 'M' | 'P' | 'R' | 'S' | 'T' | 'V' | 'W'}[,name][,pad])

An argument is entered using its initial letter. The arguments are:

A	ASSIGNS and Assigned Device
D	Device Drivers
H	Handlers
I	Interrupts
L	Libraries
M	Memory List Items
P	Ports
R	Resources
S	Semaphores
T	Tasks (Ready)
V	Volume Names
W	Waiting Tasks

If only one argument is supplied, SHOWLIST() returns a string separated by blanks. If a pad character is supplied, names will be separated by the pad rather than by blanks. If the name parameter is supplied, SHOWLIST() returns a Boolean value which indicates if the specified list contains that name. Names are case-sensitive. To provide an accurate snapshot of the current list, task switching is forbidden when the list is scanned.

```
SAY SHOWLIST('P')    → REXX MyCon
SAY SHOWLIST('P',,';')    → REXX;MyCon
SAY SHOWLIST('P','REXX')    → 1
```

STATEF() STATEF(filename)

Returns a string containing information about an external file. The string is formatted as:

```
"{DIR | FILE} length blocks protection days minutes
ticks comment."
```

The length token gives the file length in bytes, and the block token specifies the file length in blocks. For example:

```
SAY STATEF("LIBS:REXXSupport.library")
/*might give "File 2524 5 ----RW-D 4866 817 2088*/
```

WAITPKT() WAITPKT(name)

Waits for a message to be received at the specified (named) port, which must have been opened by a call to OPENPORT() within the current ARexx program. The returned Boolean value indicates whether a message packet is available at the port. Normally the returned value will be 1 (True), since the function waits until an event occurs at the message port.

The packet must then be removed by a call to GETPKT() and should be returned eventually using the REPLY() function. Any message packets received but not returned when an ARexx program exits are automatically REPLYed with the return code set to 10. For example:

```
CALL WAITPKT 'MyPort'    /*Wait awhile*/
```

Chapter 6
Debugging

ARexx provides tracing and source-level debugging facilities. Tracing displays selected statements in a program as the program exëcutes. When a clause is traced, its line number, source text, and related information are displayed on the console.

The internal interrupt system enables an ARexx program to detect certain synchronous or asynchronous events and to take special actions when they occur. Events such as a syntax error or an external halt request that would normally cause the program to exit can instead be trapped so that corrective actions can be taken.

Tracing

The tracing action selects which source clauses will be traced and has two modifier flags that control command inhibition and interactive tracing. Trace options can be shortened to one letter. The Trace options are:

ALL	All clauses are traced.
BACKGROUND	No tracing is performed and the program cannot be forced into interactive tracing.
COMMANDS	All command clauses are traced before being sent to the external host. Non-zero return codes are displayed on the console.
ERRORS	Commands that generate a non-zero return code are traced after the clause is executed.

INTERMEDIATES	All clauses are traced and intermediate results are displayed during expression evaluation. These include the values retrieved for variables, expanded compound names, and the results of function calls.
LABELS	All label clauses are traced as they are executed. A label will be displayed each time a transfer of control takes place
NORMAL (Default)	Command clauses with return codes that exceed the current error failure level are traced after execution and an error message is displayed.
OFF	Tracing is turned off.
RESULTS	All clauses are traced before execution and the final result of each expression is displayed. Values assigned to variables by ARG, PARSE or PULL instructions are also displayed. This option is recommended for general-purpose testing.
SCAN	This is a special option that traces all clauses and checks for errors, but suppresses the actual execution of the statements. It is helpful as a preliminary screening step for a newly-created program.

The tracing mode can be set using either the TRACE instruction or the TRACE() built-in function. Tracing can be selectively disabled from within a program to skip previously tested parts of a program.

Each trace line displayed on the console is indented to show the effective control (nesting) level at that clause and is identified by a special three-character code, as shown in Table 6-1. The source for each clause is preceded by its line number in the program. Expression results or intermediates are enclosed in double quotes so that leading and trailing blanks will be apparent.

Table 6-1. Special Three Character Codes

Code	Displayed Values
+++	Command or syntax error
>C>	Expanded compound name
>F>	Result of a function call
>L>	Label clause
>O>	Result of a dyadic operation
>P>	Result of a prefix operation
>U>	Uninitialized variable
>V>	Value of a variable
>>>	Expression or template result
>.>	"Place holder" token value

Tracing Output

The tracing output from a program is always directed to one of two logical streams. The interpreter first checks for a stream named STDERR and directs the output there if the stream exists. Otherwise, the trace output goes to the standard output stream STDOUT and will be interleaved with the normal console output of the program. The STDERR and STDOUT streams can be opened and closed under program control, so the programmer has complete control over the destination of tracing output.

In some cases a program may not have a predefined output stream. For example, a program invoked from a host application that did not provide input and output streams would not have an output console. To provide a tracing facility for such programs, the resident process can open a special global tracing console for use by any active program. When this console opens, the interpreter automatically opens a stream named STDERR for each ARexx program in which STDERR is not currently defined. The program then diverts its tracing output to the new stream.

A global tracing console can be opened using the TCO command utility. ARexx programs will automatically divert their tracing

output to the new window, which is opened as a standard AmigaDOS console. The user can move it and resize it as required.

The tracing console also serves as the input stream for programs during interactive tracing. When a program pauses for tracing input, the input must be entered at the trace console. Any number of programs may use the tracing console simultaneously, although it is recommended that only one program at a time be traced.

The global console can be closed using the TCC command. The closing is delayed until all read requests to the console have been satisfied. Only when all of the active programs indicate that they are no longer using the console will it actually be closed.

Command Inhibition

ARexx provides a tracing mode called command inhibition that suppresses host commands. In this mode command clauses are evaluated in the normal manner, but the command is not actually sent to the external host, and the return code is set to zero. This provides a way to test programs that issue potentially destructive commands, such as erasing files or formatting disks. Command inhibition does not apply to command clauses that are entered interactively. These commands are always performed, but the value of the special variable RC is left unchanged.

Command inhibition may be used in conjunction with any trace option. It is controlled by the "!" character, which may appear by itself or may precede any of the alphabetic options in a TRACE instruction. Each occurrence of the "!" character "toggles" the inhibition mode currently in effect. Command inhibition is cleared when tracing is set to OFF.

Interactive Tracing

Interactive tracing is a debugging facility that allows the user to enter source statements while a program is executing. These statements may be used to examine or modify variable values, issue commands, or otherwise interact with the program. Any valid language statements can be entered interactively, with the same rules and restrictions that apply to the INTERPRET instruction. In

particular, compound statements such as DO and SELECT must be complete within the entered line.

Interactive tracing can be used with any of the trace options. While in interactive tracing mode, the interpreter pauses after each traced clause and prompts for input with the code "+++". At each pause, three types of user responses are possible:

- If a null line is entered, the program continues to the next pause point.
- If a "=" character is entered, the preceding clause is executed again.
- Any other input is treated as a debugging statement and is scanned and executed.

The interpreter pauses after traceable clauses. Tracing options determine the location of the pauses. The interpreter does not pause after the instructions CALL, DO, ELSE, IF, THEN, and OTHERWISE. When any clause generates an execution error, the interpreter exits the program.

Interactive tracing is controlled by the "?" character, either by itself or in combination with an alphabetic trace option. Any number of "?" characters may precede an option. Each occurrence toggles the mode currently in effect. For example, if the current trace option was NORMAL, "TRACE ?R" would set the option to RESULTS and select interactive tracing mode. A subsequent "TRACE ?" would turn off interactive tracing.

Error Processing

The ARexx interpreter provides error processing during debugging. Errors found during interactive debugging are reported, but do not terminate the program. This special processing only applies to statements entered interactively.

ARexx also disables the internal interrupt flags during interactive debugging. This prevents an accidental transfer of control due to an error or uninitialized variable. However, if a "SIGNAL label" instruction is entered, the transfer will take place and any remaining interactive input will be abandoned. The SIGNAL instruction can still be used to alter the interrupt flags, and the new

settings will take effect when the interpreter returns to normal processing.

Each ARexx task initializes its command failure level to the client's failure level (usually 10) to suppress printing of nuisance command errors. The failure level can be changed using OPTIONS FAILAT. Command errors (RC > 0) and failures (RC >= FAILAT) can be separately trapped using SIGNAL ON ERROR and SIGNAL ON FAILURE.

The External Tracing Flag

ARexx has an external tracing flag used to force programs into interactive tracing mode. When this tracing flag is set, using the TS command utility, any program not already in interactive tracing mode will enter it immediately. The internal trace option is set to RESULTS unless it is currently set to INTERMEDIATES or SCAN, in which case it remains unchanged. Programs invoked while the external tracing flag is set will begin executing in interactive tracing mode.

The external tracing flag provides a way to regain control over looping or unresponsive programs. Once a program enters interactive tracing mode, the user can step through the program statements and diagnose the problem. External tracing is a global flag, so all currently-active programs are affected by it. The tracing flag remains set until it is cleared using the TE command utility. Each program maintains an internal copy of the last state of the tracing flag and sets its tracing option to OFF when it observes that the tracing flag has been cleared. Programs in BACKGROUND tracing mode do not respond to the external tracing flag.

Interrupts

ARexx maintains an internal interrupt system used to detect and trap certain error conditions. When an interrupt is enabled and its corresponding condition arises, a transfer of control to the label specific to that interrupt occurs. This allows a program to retain control in circumstances that might otherwise cause the program to terminate. The interrupt conditions can be caused by either

synchronous events, like a syntax error, or asynchronous events, like a Ctrl+C break request.

Note These internal interrupts are completely separate from the hardware interrupt system managed by the EXEC operating system.

The name assigned to each interrupt is actually the label to which control will be transferred. Thus, a SYNTAX interrupt will transfer control to the label "SYNTAX:". Interrupts can be enabled or disabled using the SIGNAL instruction. For example, the instruction "SIGNAL ON SYNTAX" would enable the SYNTAX interrupt.

The interrupts supported by ARexx are:

BREAK_C This traps (detects and treats as a signal and not as normal output) a Ctrl+C break request generated by AmigaDOS. If the interrupt is not enabled, the program terminates immediately with the error message "Execution halted" and returns with the error code set to 2.

BREAK_D This traps a Ctrl+D break request issued by AmigaDOS. The break request is ignored if the interrupt is not enabled.

BREAK_E This traps a Ctrl+E break request issued by AmigaDOS. The break request is ignored if the interrupt is not enabled.

BREAK_F This traps a Ctrl+F break request issued by AmigaDOS. The break request is ignored if the interrupt is not enabled.

ERROR This interrupt is generated by any host command that returns a non-zero code.

HALT An external halt request is trapped if this interrupt is enabled. Otherwise, the program terminates immediately with the error message "Execution halted".

IOERR Errors detected by the I/O system are trapped if this interrupt is enabled.

NOVALUE An interrupt occurs if an uninitialized variable is used while this condition is enabled. The usage could be within an expression, in the UPPER instruction, or with the VALUE() built-in function.

SYNTAX A syntax or execution error is generated by this interrupt. Not all such errors can be trapped. Certain errors occur before a program is executed and those detected by the ARexx external interface cannot be trapped by the SYNTAX interrupt.

When an interrupt forces a transfer of control, all of the currently active control ranges are dismantled and the interrupt that caused the transfer is disabled. This disabling prevents a possible recursive interrupt loop. Only the control structures in the current environment are affected, so an interrupt generated within a function will not affect the caller's environment.

Two special variables are affected when an interrupt occurs:

SIGL Always set to the current line number before the transfer of control takes place. This allows the determination of which source line is executed.

RC Set to the error code that caused the condition. For ERROR interrupts, this value will be a command return code and can usually be interpreted as an error severity level. The value for SYNTAX interrupts is always an ARexx error code.

Interrupts are useful for error-recovery actions. This involves informing external programs that an error occurred or reporting further diagnostics to isolate the problem. Program 15 issues a "message" command to an external host called "MyEdit" whenever a syntax error is detected.

Program 15. Interrupt.rexx

```
/*A macro program for 'MyEdit'*/
SIGNAL ON SYNTAX          /*Enable interrupt*/
(normal processing)
EXIT
SYNTAX:                 /*Syntax error detected*/
ADDRESS 'MyEdit'
'message' 'error' RC errortext(RC)
EXIT 10
```

Chapter 7
Parsing

Parsing extracts substrings from a string and assigns them to variables. Parsing is performed using the PARSE instruction or its variants ARG and PULL. The operation input is called the parse string and comes from several sources, including argument strings, expressions, or the console.

String-manipulation functions like SUBSTR() and INDEX() may be used for parsing, but the PARSE instruction statement is more efficient, especially if extracting many fields from a string.

Templates

Parsing is controlled by a template, a group of tokens that specifies both the variables to be given values and the way to determine the value strings. The way tokens are arranged in the template determines whether the token is one of two basic template objects: a marker or a target.

Marker	Determines the starting and ending position in the parse string or the scan position.
Target	A symbol assigned a value by the parsing operation. That value is the substring determined by the marker positions.

Markers

There are three types of marker objects:

Absolute markers	Actual index position in the parse string.
Relative markers	A positive or negative offset from the current position.
Pattern markers	Matches the pattern against the parse string beginning at the current scan position.

Targets

Targets, like markers, can affect the scan position if value strings are being extracted by tokenization. Parsing by tokenization extracts words (tokens) from the parse string and is used whenever a target is followed immediately by another target. During tokenization the current scan position is advanced past any blanks to the start of the next word. The ending index is the position just past the end of the word and the value string has neither leading nor trailing blanks.

Targets are specified by variable symbols. The place holder, denoted by a period (.), is a special type of target and behaves like a normal target except that it does not have an assigned value.

Template Objects

Each template object is specified by one or more tokens:

Symbols	A symbol may specify a target or a marker. It's a marker if it follows an operator (+, - or =) and the symbol value is used as an absolute or relative position. Symbols enclosed in parentheses specify pattern markers, and the symbol value is used as the pattern string. It specifies a target if neither of the preceding cases apply and the symbol is variable. Fixed symbols always specify absolute markers and must be whole numbers. The only exception is the place holder (.) target.
Strings	A string always represents a pattern marker.

Parentheses A symbol enclosed in parentheses is a pattern marker and the value of the symbol is used as the pattern string. While the symbol may be either fixed or variable, it will usually be a variable. A fixed pattern could be given more simply as a string.

Operators The three operators (+, – and =) are valid within a template and must be followed by a fixed or variable symbol. The value of the symbol is used as a marker and must represent a whole number. The "+" and "–" operators signify a relative marker, whose value is negated by the "–" operator. The "=" operator indicates an absolute marker and is optional if the marker is defined by a fixed symbol.

Commas The comma (,) marks the end of a template. It is also used as a separator when multiple templates are provided with an instruction. The interpreter obtains a new parse string before processing each succeeding template. For some source options, the new string will be identical to the previous one. The ARG, EXTERNAL and PULL options will generally supply a different string, as will the VAR option if the variable has been modified.

The ARexx interface command parser has been generalized to recognize double-delimiter sequences within a (quoted) string file. The quoting convention is convenient for short programs, but it is easy to run out of quoting levels in longer programs. Single and double-quotes within a REXX program are equivalent, but the external environment may make a distinction.

AmigaDOS uses double-quotes. Strings entered from a Shell must begin with a double-quote, especially if you wish to include semicolons. For example:

```
RX "SAY 'It''s possible, indeed; you ain''t seen
nothin'' yet!' "
→ It's possible, indeed; you ain't seen nothin' yet!

RX "SAY '""Hello!""'"     → "Hello!"
```

The Scanning Process

Scan positions are expressed as an index in the parse string and can range from 1 (the start of the string) to the length of the string plus 1 (the end).

The substring specified by two scan indices includes the characters from the starting position up to, but not including, the ending position. For example, the indices 1 and 10 specify characters 1-9 in the parse string. If the second scan index is less than or equal to the first, the remainder of the parse string is used as the substring. This means that a template specification like:

```
PARSE ARG 1 all 1 first second
```

will assign the entire parse string to the variable ALL. If the current scan index is already at the end of the parse string, the remainder is the null string.

When a pattern marker is matched against the parse string, the marker position is the index of the first character of the matched pattern or the end of the string if no match was found. The pattern is removed from the string whenever a match is found. This is the only operation that modifies the parse string during the parsing process.

Templates are scanned from left to right with the initial scan index set to 1. The scan position is updated each time a marker object is encountered, according to the type and value of the marker. Whenever a target object is found, the assigned value is determined by examining the next template object. If the next object is another target, the value string is determined by tokenizing the parse string. Otherwise, the current scan position is used as the start of the value string and the position specified by the following marker is used as the end point.

The scan continues until all of the objects in the template have been used. Every target will be assigned a value. Once the parse string has been exhausted, the null string is assigned to any remaining targets.

Parsing Examples

Parsing by Tokenization

Computer programs frequently split a string into its component words or tokens. This is accomplished with a template consisting entirely of variables (targets).

```
/*Assume "hammer 1 each $600.00" was entered*/
PULL item qty units cost .
```

In this example the input line from the PULL instruction is split into words and assigned to the variables in the template. The variable item receives the value "hammer", qty is set to "1", units is set to "each" and cost gets the value "$600.00". The final place holder (.) is given a null value, since there are only four words in the input. However, it forces the preceding variable cost to be given a tokenized value. If the place holder were omitted, the remainder of the parse string would be assigned to cost, which would then have a leading blank.

```
answer = "Only Amiga makes it possible."
DO forever
   PARSE VAR answer first answer
/*Place first word into 'first' and the rest into
'answer'.*/
   IF first ='' THEN LEAVE
   /*Stop if there are no more words*/
   SAY answer
   END
```

The first word of a string is removed and the remainder is placed back in the string. The process continues until no more words are extracted. The output is:

```
Amiga makes it possible.
makes it possible.
it possible.
possible.
```

Parsing by Pattern

Pattern markers extract the desired fields. The "pattern" in this case is very simple — a single character — but could be an arbitrary string of any length. This form of parsing is useful whenever delimiter characters are present in the parse string.

```
/*Assume an argument string "12,35.5,1"  */
ARG hours ',' rate ',' withhold
```

The pattern is actually removed from the parse string when a match is found. If the parse string is scanned again from the beginning, the length and structure of the string may be different than at the start of the parsing process. The original source of the string, however, is never modified.

Parsing by Positional Markers

Parsing with positional markers is used whenever the files of interest are known to be in certain positions in a string.

```
/* Records look like: */
/* Start:  1-5 */
/* Length: 6-10 */
/* Name: @ (start,length)*/
PARSE value record with 1 start +5 length +5 =start
name +length
```

The records being processed contain a variable length field. The starting position and length of the field are given in the first part of the record with a variable positional marker used to extract the desired field.

The "=start" sequence is an absolute marker whose value is the position placed in the variable start earlier in the scan. The "+length" sequence supplies the effective length of the field.

Multiple Templates

More than one template can be specified with an instruction by separating the templates with a comma. The ARG instruction (or PARSE UPPER ARG) accesses the argument strings provided when the program was called. Each template accesses the succeeding argument string. For example:

```
/*Assume arguments are ('one two',12,sort)*/
ARG first second,amount,action,option
```

The first template consists of the variables first and second, which are set to the values "one" and "two". In the next two templates, amount gets the value "12" and action is set to "SORT". The last template consists of the variable "option", which is set to the null string, since only three arguments were available.

When multiple templates are used with the EXTERNAL or PULL source options, each additional template requests an additional line of input from the user:

```
/*Read last, first, and middle names and ssn*/
PULL last ',' first middle,ssn
```

Two lines of input are read. The first input line is expected to have three words which are assigned to the variables "last", "first", and "middle". The first variable is followed by a comma. The entire second input line is assigned to the variable "ssn".

Multiple templates can be useful even with a source option that returns the identical parse string. If the first template included pattern markers that altered the parse string, the subsequent templates could still access the original string. Subsequent parse strings obtained from the VALUE source do not cause the expression to be re-evaluated, but only retrieve the prior result.

Appendix A
Error Messages

When the ARexx interpreter detects an error in a program, it returns an error code to indicate the nature of the problem. Errors are normally handled by displaying the error code, the source line number where the error occurred and a brief message explaining the error condition. Unless the SYNTAX interrupt has been previously enabled (using the SIGNAL instruction), the program then terminates and control returns to the caller. Most syntax and execution errors can be trapped by the SYNTAX interrupt, allowing the user to retain and perform whatever special error processing is required. Certain errors are generated outside of the context of an ARexx program and therefore cannot be trapped by this mechanism. Refer to Chapter 6 for further information on error trapping and processing.

Each error code is associated with a severity level that is reported to the calling program as the primary result code. The values of these results codes are 5 (least serious), 10 (moderately serious), and 20 (very serious). The error code itself is returned as the secondary result. The subsequent propagation or reporting of these codes is dependent on the external (calling) program.

The following pages list all of the currently-defined error codes, along with the associated result code and message strings.

Table A-1. Error Codes and Messages

Error	Result Code	Message	Explanation
1	5	Program not found	The named program could not be found or was not an ARexx program. ARexx programs are expected to start with a comment (/*...*/). This error is detected by the external interface and cannot be trapped by the SYNTAX interrupt.
2	10	Execution halted	A Ctrl+C break or an external halt request was received and the program terminated. This error will be trapped if the HALT interrupt has been enabled.
3	20	Insufficient memory	The interpreter was unable to allocate enough memory for an operation. Since memory space is required for all parsing and execution operations, this error cannot usually be trapped by the SYNTAX interrupt.
4	10	Invalid character	A non-ASCII character was found in the program. Control codes and other non-ASCII characters may be used in a program by defining them as hex or binary strings. This is a scan-phase error and cannot be trapped by the SYNTAX interrupt.
5	10	Unmatched quote	A closing single or double quote was missing. Check that each string is properly delimited. This is a scan-phase error and cannot be trapped by the SYNTAX interrupt.

Error	Result Code	Message	Explanation
6	10	Unterminated comment	The closing */ of a comment was not found. Remember that comments may be nested, so each /* must be matched by a */. This is a scan-phase error and cannot be trapped by the SYNTAX interrupt.
7	10	Clause too long	A clause was too long for the internal buffer. The source line should be broken into smaller parts. This is a scan-phase error and cannot be trapped by the SYNTAX interrupt.
8	10	Invalid token	An unrecognized lexical token was found, or a clause could not be properly classified. This is a scan-phase error and cannot be trapped by the SYNTAX interrupt.
9	10	Symbol or string too long	An attempt was made to create a string longer than the maximum allowed.
10	10	Invalid message packet	An invalid action code was found in a message packet sent to the ARexx resident process. The packet was returned without being processed. This error is detected by the external interface and cannot be trapped by the SYNTAX interrupt.
11	10	Command string error	A command string could not be processed. This error is detected by the external interface and cannot be trapped by the SYNTAX interrupt.
12	10	Error return from function	An external function returned a non-zero error code. Check that the correct parameters were supplied to the function.

Error	Result Code	Message	Explanation
13	10	Host environment not found	The message port corresponding to a host address string could not be found. Check that the required external host is active.
14	10	Requested library not found	An attempt was made to open a function library included in the Library List, but the library could not be opened. Check that the correct name and version of the library were specified when the library was added to the resource list.
15	10	Function not found	A function was called that could not be found in any of the currently accessible libraries and could not be located as an external program. Check that the appropriate function libraries have been added to the Libraries List.
16	10	Function did not return value	A function was called which failed to return a result string, but did not otherwise report an error. Check that the function was programmed correctly or invoke it using the CALL instruction.
17	10	Wrong number of arguments	A call was made to a function which expected a different number of arguments. This error will be generated if a built-in or external function is called with more arguments than can be accommodated in the message packet used for external communications.
18	10	Invalid argument to function	An inappropriate argument was supplied to a function or a required argument was missing. Check the parameter requirements specified for the function.

Error	Result Code	Message	Explanation
19	10	Invalid PROCED-URE	A PROCEDURE instruction was issued in an invalid context. Either no internal functions were active or a PROCEDURE had already been issued in the current storage environment.
20	10	Unexpected THEN or WHEN	A WHEN or THEN instruction was executed outside of a valid context. The WHEN instruction is valid only within a SELECT range, and THEN must be the next instruction following an IF or WHEN.
21	10	Unexpected ELSE or OTHER-WISE	An ELSE or OTHERWISE was found outside of a valid context. The OTHERWISE instruction is valid only within a SELECT range. ELSE is valid only following the THEN branch of an IF range.
22	10	Unexpected BREAK, LEAVE or ITERATE	The BREAK instruction is valid only within a DO range or inside an INTERPRETed string. The LEAVE and ITERATE instructions are valid only within an iterative DO range.
23	10	Invalid statement in SELECT	An invalid statement was encountered within a SELECT range. Only WHEN, THEN and OTHERWISE statements are valid within a SELECT range, except for the conditional statements following THEN or OTHERWISE clauses.
24	10	Missing or multiple THEN	An expected THEN clause was not found or another THEN was found after one had already been executed.
25	10	Missing OTHER-WISE	None of the WHEN clauses in a SELECT succeeded, but no OTHERWISE clause was supplied.

Error	Result Code	Message	Explanation
26	10	Missing or unexpected END	The program source ended before an END was found for a DO or SELECT instruction, or an END was encountered outside a DO or SELECT range.
27	10	Symbol mismatch	The symbol specified on an END, ITERATE or LEAVE instruction did not match the index variable for the DO range. Check that the active loops have been nested properly.
28	10	Invalid DO syntax	An invalid DO instruction was executed. An initializer expression must be given if a TO or BY expression is specified. A FOR expression must yield a non-negative integer result
29	10	Incomplete IF or SELECT	An IF or SELECT range ended before all of the required statements were found. Check whether the conditional statement following a THEN, ELSE or OTHERWISE clause was omitted.
30	10	Label not found	A label specified by a SIGNAL instruction or implicitly referenced by a enabled interrupt could not be found in the program source. Labels defined dynamically by an INTERPRET instruction or by interactive input are not included in the search.
31	10	Symbol expected	A non-symbol token was found where only a symbol token is valid. The DROP, END, LEAVE, ITERATE and UPPER instructions may only be followed by a symbol token. This message will also be issued if a required symbol is missing.

Error	Result Code	Message	Explanation
32	10	Symbol or string expected	An invalid token was found in a context where only a symbol or string is valid.
33	10	Invalid keyword	A symbol token in an instruction clause was identified as a keyword, but was invalid in the specific context.
34	10	Required keyword missing	An instruction clause required a specific keyword token to be present, but it was not supplied. This message will be issued if a SIGNAL ON instruction is not followed by one of the interrupt keywords (e.g., SYNTAX).
35	10	Extraneous characters	A seemingly valid statement was executed, but extra characters were found at the end of the clause.
36	10	Keyword conflict	Two mutually exclusive keywords were included in an instruction clause, or a keyword was included twice in the same instruction.
37	10	Invalid template	The template provided with an ARG, PARSE or PULL instruction was not properly constructed.
38	10	Invalid TRACE request	The alphabetic keyword supplied with a TRACE instruction or as the argument to the TRACE() built-in function was not valid.
39	10	Uninitialized variable	An attempt was made to use an uninitialized variable while the NOVALUE interrupt was enabled.
40	10	Invalid variable name	An attempt was made to assign a value to a fixed symbol.

Error	Result Code	Message	Explanation
41	10	Invalid expression	An error was detected during the evaluation of an expression. Check that each operator has the correct number of operands and that no extraneous tokens appear in the expression. This error will be detected only in expressions that are actually evaluated. No checking is performed on expressions in clauses that are being skipped.
42	10	Unbalanced parentheses	An expression was found with an unequal number of opening and closing parentheses.
43	10	Nesting limit exceeded	The number of subexpressions in an expression was greater than the maximum allowed. Simplify the expression by breaking it into two or more intermediate expressions.
44	10	Invalid expression result	The result of an expression was not valid within its context. This message will be issued if an increment or limit expression in a DO instruction yields a non-numeric result.
45	10	Expression required	An expression was omitted in a context where one is required. For example, the SIGNAL instruction, if not followed by the keywords ON or OFF, must be followed by an expression.
46	10	Boolean value not 0 or 1	An expression result was expected to yield a boolean result, but evaluated to something other than 0 or 1.

Error	Result Code	Message	Explanation
47	10	Arithmetic conversion error	A non-numeric operand was used in an operation requiring numeric operands. This message will also be generated by an invalid hex or binary string.
48	10	Invalid operand	An operand was not valid for the intended operation. This message will be generated if an attempt is made to divide by 0 or if a fractional exponent is used in an exponentiation operation.

Appendix B
Command Utilities

ARexx provides a number of command utilities, located in the REXXC Directory, that provide various control functions. These are executable modules that can be run from the Shell and are relevant only when the ARexx resident process is active.

HI HI

Sets the global halt flag, which causes all active ARexx programs to receive an external halt request. Each program will exit immediately unless its HALT interrupt has been enabled. The halt flag does not remain set, but is cleared automatically after all current programs have received the request.

RX RX name [arguments]

Launches an ARexx program. If the specified name includes an explicit path, only that directory is searched for the program; otherwise, the current directory and REXX: are checked for a program with the given name. The optional argument string is passed to the program.

RXSET RXSET [name [[=] value]]

Adds a (name,value) pair to the Clip List. Name strings are assumed to be in mixed case. If a pair with the same name already exists, its value is replaced with the current string. If a name without a value string is given, the entry is removed from the Clip List. If RXSET is invoked without arguments, it will list all (name, value) pairs in the Clip List.

RXC RXC

Closes the resident process. The "REXX" public port is withdrawn immediately, and the resident process exits as soon as the last ARexx program finishes. No new programs can be launched after a close request.

TCC TCC

Closes the global tracing console as soon as all active programs are no longer using it. All read requests queued to the console must be satisfied before it can be closed.

TCO TCO

Opens the global tracing console. The tracing output from all active programs is diverted automatically to the new console. The console window can be moved and resized by the user and can be closed with the TCC command.

TE TE

Clears the global tracing flag, which forces the tracing mode to OFF for all active ARexx programs.

TS TS

Starts interactive tracing by setting the external trace flag, which forces all active ARexx programs into interactive tracing mode. Programs will start producing trace output and will pause after the next statement. This command is useful for regaining control over programs caught in infinite loops or otherwise misbehaving. The trace flag remains set until cleared by the TE command, so subsequent program invocations will be executed in interactive tracing mode.

WaitForPort WaitForPort [name of port]

WaitForPort waits 10 seconds for the specified port to appear. A return code of 0 indicates that the port was found. A return code of 5 indicates that the application is not currently running or that the port does not exist. Port names are case sensitive. For example:

```
WaitForPort ED_1
WaitForPort MyPort
```

Glossary

This glossary provides definitions of terms used in the ARexx manual.

address
An identifying number assigned to every byte of information in a computer's memory and every sector on a disk.

argument
An additional piece of information included with an instruction or function. The argument determines the action to be taken.

assignment clause
A variable symbol (simple, stem or compound symbol) followed by an = operator. In an assignment clause, the tokens to the right of the = sign are evaluated and the result is assigned to the variable symbol.

boolean

Having two possible states: 0 (false) or 1 (true).

clause

The smallest executable language unit.

clip list

A clipboard used for intertask communication. Names and value pairs can be added to the clip list with the SETCLIP() function.

command clause

An ARexx expression in which the result is issued as a command to an external application.

command interface

A message port through which ARexx issues commands to compatible applications.

comment

A group of characters enclosed in /*....*/ symbols. Each ARexx program starts with a comment.

compound symbol

A token made up of alphanumeric or ., !, ?, $, _ characters with one or more periods within the name. Compound symbols have the structure stem.n1.n2...nk.

concatenation

The process of linking together two strings.

debugging

Finding and fixing errors in a program.

delimiter

A character that marks the beginning and end of a string. ARexx delimiters are a single quote (') and a double quote (").

expression

A group of tokens to be evaluated. Expressions are made up of strings, symbols, operators, and parentheses.

fixed symbol

A token beginning with a digit or a period.

function

A group of statements that can be executed as a whole. Functions allow you to build complex programs from smaller modules.

function hosts

An external application that contains an ARexx port. The name of a function host is the name of the program's public message port.

function libraries

A collection of one or more functions organized as an Amiga shared library. A function library must contain a library name, a search priority, an entry point offset, and a version number.

global environment

The fixed components of an ARexx program, including the program source code, static data strings, and argument strings.

host address

In an outside application, the host addess is the name of the message port to which ARexx commands can be sent.

instruction

A clause beginning with a certain keyword that tells ARexx to perform an action.

interprocess communication

The exchange of information between applications.

interrupts

Internal flags in ARexx that allow a program to detect errors and retain control when the program would normally abort. Interrupts are controlled by the SIGNAL: instruction.

iteration

Repeating a section of a program.

label

A symbol token followed by a colon (:). Labels identify a particular location in the program.

library list

An internal list, maintained by ARexx, of all the currently available function libraries and function hosts. Applications can add or remove function libraries as needed.

macro

Another name for an ARexx program.

marker

A token that determines the beginning and end of a parse string.

message port

An interface in an Amiga application that allows the program to communicate with an ARexx program.

null clause

A blank line or a comment line.

numeric precision

The number of decimal places in an arithmetic result. As the number of decimal places decreases, the result becomes less precise.

operators

A character such as (+), (–), or (|) used in an arithmetic, concatenation, comparison, or logical operation.

parsing

Breaking a string into smaller units.

return code

The severity level of an error. This number (5, 10, or 20) is displayed when an error occurs in an ARexx program.

RexxMast

The program that acts as an interpreter for ARexx programs.

simple symbol

A token that does not begin with a digit or contain any periods.

statement

An assignment, instruction, or command clause.

stem symbol

A token that has one period at the end of its name. Stem symbols are used to initialize compound symbols.

storage environment

The variable components of an ARexx program, including the symbol table, numeric options, trace options, and host address strings.

string

A group of characters beginning and ending with a delimiter (single or double quote). The value of a string is the string itself.

symbol

Any group of the alphanumeric characters a-z, A-Z, 0-9, period (.), exclamation point (!), question mark (?), dollar sign ($), or underscore (_).

symbol table

An internal table created by ARexx that stores the value strings that have been assigned to the variables in a program.

target

A symbol, usually a variable symbol, that is assigned a value during a parsing operation.

template

A group of tokens that specifies the variables used in a parsing operation. It also specifies the way in which the values will be determined.

tokenization

The process of breaking a statement into its individual tokens.

tokens

The smallest entities of the ARexx language.

tracing

Displaying the lines of an ARexx program as the program is executing. This allows you to determine exactly where any errors are occurring.

variable

A symbol that can be assigned a value.

Index

A

ABBREV(), 5-8
ABS(), 5-8
ADDLIB(), 5-8
ADDRESS, 3-18, 4-2, 4-19
ADDRESS(), 3-18, 4-2, 5-9
ALLOCMEM(), 5-19, 5-39
ARG, 2-8, 4-3, 6-2, 7-1
ARG(), 5-9
argument strings, 2-7, 3-24, 4-3, 7-1
assignment clauses, 3-14

B

B2C(), 5-9
BITAND(), 5-10
BITCHG(), 5-10
BITCLR(), 5-10
BITCOMP(), 5-11
BITOR(), 5-11
BITSET(), 5-11
BITTST(), 5-11
BITXOR(), 5-12
blanks, 3-7
BREAK, 4-4, 4-9
built-in functions, 5-3, 5-5

C

C2B(), 5-12
C2D(), 5-12
C2X(), 5-12
CALL, 4-2, 4-4, 5-2
CENTER(), 5-13
CENTRE(), 5-13
clauses, 3-1, 3-13
    assignment, 3-14
    command, 3-13, 3-15, 6-2, 6-4
    continuation of, 3-13
    instruction, 3-13, 3-15
    label, 3-14, 4-9, 5-2, 6-2
    null, 3-14
Clip List, 5-6
    adding values, 5-26
    examining, 5-27
    searching, 5-19
CLOSE(), 5-13
CLOSEPORT(), 5-39
command
    clauses, 3-13, 3-15, 6-2
    interface, 3-1, 3-18
    shell, 3-22, 4-17
command clauses, 6-4
comment line, 2-4
comments, 3-2, 3-14
COMPARE(), 5-13
compound symbols, 3-3, 3-25, 4-7, 4-16
COMPRESS(), 5-13

conditional statements, 4-7, 4-9, 4-11, 4-19, 4-21
converting
    binary digits, 5-9
    characters, 5-12
    decimal to hexadecimal, 5-14
    hexadecimal digits, 5-34
    hexadecimal to decimal, 5-34
    to decimal, 5-12
    to hexadecimal, 5-12
COPIES(), 5-14

D

D2C(), 5-14
D2X(), 5-14
DATATYPE(), 5-14
DATE(), 5-15
debugging, 6-1
delimiter, 3-5
DELSTR(), 5-16
DELWORD(), 5-16
DIGITS(), 5-16
displaying output, 4-19
DO, 2-6, 4-4, 4-8, 4-10, 6-5
DROP, 4-7

E

ECHO, 4-7
ELSE, 4-7
END, 4-4, 4-8, 4-10, 4-19
engineering notation, 4-11
environment, 3-23
    external, 3-24
    global, 3-24
    internal, 3-24
    storage, 3-24, 4-2, 5-2, 5-5
EOF(), 5-17
error checking
    tracing, 1-3, 2-8, 6-1
error processing, 6-5
ERRORTEXT(), 5-17
EXISTS(), 5-17
EXIT, 2-8, 4-8, 4-19
EXPORT(), 5-17
expressions, 3-1, 3-15
external
    environment, 3-24
    files, 5-3
        opening, 5-22
    function libraries, 5-3
    tracing flag, 6-6

F

FIND(), 5-18
fixed symbols, 3-16, 7-2
FORM(), 5-18
FREEMEM(), 5-40
FREESPACE(), 5-18
function hosts, 5-4, 5-5
function libraries, 1-3, 5-4, 5-5
functions, 2-7, 5-1
    built-in, 5-3, 5-5
    internal, 3-24, 5-2, 5-5
    invoking, 4-4, 5-1
FUZZ(), 5-19

G

GETARG(), 5-40
GETCLIP(), 5-6, 5-19
GETPKT(), 5-40
GETSPACE(), 5-19
global
    environment, 3-24
    tracing console, 6-3

H

HALT, 4-20
HASH(), 5-19
HI, B-1
host address, 3-18, 3-24, 4-2, 5-9

I

IF, 2-7, 4-9
IMPORT(), 5-18, 5-20
INDEX(), 5-20, 7-1
initializer expression, 4-5
input stream, 3-22, 3-24, 6-4
INSERT(), 5-20
instruction clauses, 3-13, 3-15
instructions, 2-5, 4-1
interactive tracing, 6-1, 6-4
internal
    defaults, 4-12
    environment, 3-24
    functions, 5-2, 5-5
    interrupt flags, 4-20
    interrupts, 6-1, 6-5
INTERPRET, 4-4, 4-9, 6-4
interprocess communication, 1-2
interrupt flags, 4-20, 5-2, 6-5, 6-7
interrupts, 6-6
invoking functions, 4-4
ITERATE, 4-9, 4-10

K

keywords, 4-1

L

label clauses, 3-14, 4-9, 5-2, 6-2
LASTPOS(), 5-20
LEAVE, 4-4, 4-9, 4-10
LEFT(), 5-21
LENGTH(), 5-21
libraries
    function, 5-3
    shared, 5-3
library list, 5-4
    adding libraries, 5-8
    examining, 5-27
    removing entries, 5-25
LINES(), 5-21
loops, 4-5

M

macros, 3-20
markers, 7-1, 7-2
    positional, 7-6
MAX(), 5-22
memory
    allocating blocks, 5-19, 5-39
    releasing blocks, 5-40
message ports
    checking, 5-40
    closing, 5-39
    creating, 5-41
MIN(), 5-22
multiple templates, 4-14, 7-3, 7-7
multitasking, 1-2

N

NOP, 4-7, 4-11
null clauses, 3-14
NUMERIC, 3-8, 4-11, 5-2

Numeric Digits, 3-8, 3-10, 4-11, 5-16
    engineering notation, 4-11
    scientific notation, 4-11

O

OPEN(), 5-22
OPENPORT(), 5-41
operators, 3-6, 3-16, 7-3
    arithmetic, 3-7
    comparison, 3-7, 3-10
    concatenation, 3-7, 3-9
    logical, 3-7, 3-11
OPTIONS, 4-12, 5-2, 6-6
OTHERWISE, 4-12
output stream, 3-24, 6-3
OVERLAY(), 5-22

P

parentheses, 3-16, 7-3
PARSE, 4-13, 6-2, 7-1
parsing, 7-1
    by pattern, 7-6
    by tokenization, 7-2, 7-5
    input sources, 4-14
    marker, 7-1, 7-2
    multiple templates, 7-3, 7-7
    targets, 7-1, 7-2, 7-4
    templates, 7-1
    the scanning process, 7-4
pattern
    marker, 7-4
    markers, 7-2
    parsing, 7-6
    searching, 5-23
POS(), 5-23
positional markers, 7-6
PRAGMA(), 5-23
PROCEDURE, 4-16, 5-2
PULL, 2-6, 4-17, 6-2, 7-1
PUSH, 4-17

Q

QUEUE, 4-18

R

random numbers, 5-24, 5-25
RANDOM(), 5-24
RANDU(), 5-25
RC variable, 3-22, 6-8
READCH(), 5-25
READLN(), 5-25
REMLIB(), 5-25
REPLY(), 5-41
RESULT variable, 4-4
RETURN, 2-8, 4-18, 5-3
return codes, 3-22, 4-12, 6-1, 6-2
REVERSE(), 5-26
REXX:, 2-2, 2-3
RexxMast, 2-1, 2-4, 3-23
REXXSupport.libary
    opening, 5-38
REXXSupport.library, 5-38
RIGHT(), 5-26
running programs, 2-6
RX, 2-3, B-1
RXC, B-2
RXSET, B-1

S

SAY, 2-6, 4-7, 4-19

scan position, 7-1, 7-2, 7-4
scientific notation, 4-11
SEEK(), 5-26
SELECT, 4-8, 4-19, 6-5
SETCLIP(), 5-26
SHELL, 4-19
SHOW(), 5-9, 5-27
SHOWDIR(), 5-41
SHOWLIST(), 5-42
SIGL variable, 4-21, 6-8
SIGN(), 5-27
SIGNAL, 4-20, 5-2, 6-5, 6-7
signal conditions, 4-20
simple symbols, 3-3, 3-25
SOURCELINE(), 5-27
SPACE(), 5-28
STATEF(), 5-43
STDERR, 4-14, 6-3
STDIN, 4-17, 4-18
STDOUT, 6-3
stem symbols, 3-3, 3-25, 4-16
storage environment, 3-24, 4-2, 5-2, 5-5
STORAGE(), 5-18, 5-28
strings, 2-5, 3-5, 3-14, 3-16, 7-2
    comparing, 5-11, 5-13
    parsing, 4-13
    removing blanks, 5-29
    reversing characters, 5-26
STRIP(), 5-29
SUBSTR(), 5-29, 7-1
SUBWORD(), 5-29
symbol table, 3-16, 3-24, 4-16, 5-2
SYMBOL(), 5-30
symbols, 2-5, 3-2, 3-16, 7-2
    compound, 3-3, 3-25, 4-7, 4-16
    fixed, 3-16, 7-2
    simple, 3-3, 3-25
    stem, 3-3, 3-25, 4-16
SYNTAX, 6-7

T

targets, 7-1, 7-2, 7-4
TCC, 6-4, B-2
TCO, 6-3, B-2
TE, 6-6, B-2
template, 4-3, 4-13, 4-17, 7-1
    multiple, 4-14, 7-3, 7-7
    objects, 7-1, 7-2
TIME(), 5-30
tokenization, 3-24, 7-5
tokens, 3-1
TRACE, 2-8, 6-2
TRACE(), 5-31, 6-2
tracing, 1-3, 5-31, 6-1
    display formatting, 6-2
    global tracing console, 6-3
    interactive, 6-1, 6-4
    interrupts, 6-6
    options, 3-24, 5-2, 6-1, 6-5, 6-6
    output, 6-3
TRANSLATE(), 5-31
TRIM(), 5-31
TRUNC(), 5-32
TS, 6-6, B-2

U

UPPER(), 5-32

V

VALUE(), 5-32
variables, 2-6, 3-3, 3-7, 3-24, 4-13, 4-16, 7-1
    resetting values, 4-7
VERIFY(), 5-32

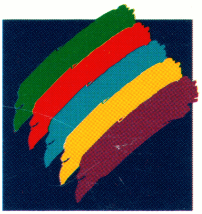
W

WaitForPort, B-3
WAITPKT(), 5-43
WHEN, 4-21
WORD(), 5-33
WORDINDEX(), 5-33
WORDLENGTH(), 5-33
WORDS(), 5-33
WRITECH(), 5-33
WRITELN(), 5-34

X

X2C(), 5-34
X2D(), 5-34
XRANGE(), 5-34

AMIGA



Commodore

371162-01

This was brought to you

from the archives of

http://retro-commodore.eu